Australian Government
Department of Defence
Science and Technology

Conversion of DST Group Shape Optimisation Software for Increased Portability across Computing Platforms

Robert Kaye and Witold Waldman

Aerospace Division
Defence Science and Technology Group

DST-Group-TR-3251

ABSTRACT

The DST Group shape optimisation methodology is well established with several successful implementations to ADF aircraft involving repairs to crack-prone locations. The process involves adaptive reshaping of locally-concave boundaries so as to minimise a stress concentration by spreading the stress more evenly over a longer region of the boundary. DST Group in-house software is used in conjunction with a commercial finite element solver in an iterative manner to achieve this outcome. The prior DST Group software had been found to be dependent on the version of the commercial graphical user interface being used and the software was not readily adaptable to newer versions. The work reported here involves replacing some of the prior code so that the graphical user interface is not used in the process. This document includes a number of 2D and 3D example problems that were used to demonstrate the successful operation of the converted code. The main benefit of the new code is that the software can be ported to other computer hardware without any interaction with the installed graphical user interface, but it also enables a reduction in the use of commercial licenses and provides faster run times.

RELEASE LIMITATION

Approved for public release

Published by

Aerospace Division
Defence Science and Technology Group
506 Lorimer Street
Fishermans Bend, Victoria 3207, Australia

Telephone: 1300 333 362
Fax: (03) 9626 7999

AR-016-589
May 2016

APPROVED FOR PUBLIC RELEASE

Conversion of DST Group Shape Optimisation Software for Increased Portability across Computing Platforms

Executive Summary

Over the past sixteen years, the Aerospace Division of DST Group has developed and implemented a technology of computer-based reshaping of aircraft structural details that alleviates stress concentrations and provides much improved fatigue life. This reshaping can include the removal of an existing fatigue crack if necessary. Notable implementations have increased the fatigue life of the F-111 wing pivot fitting structure and more recently the F/A-18 LAU-7 missile launcher guide rail. The methodology involves a combination of DST Group in-house shape optimisation software and commercial finite element analysis software used under license. As newer versions of the commercial software have come into use, a variety of compatibility problems with the DST Group software have become apparent. As a result of such issues, it is now necessary to make modifications to the process so that it is less reliant on the commercial software, while at the same time eliminating the compatibility problems.

The modifications presented in this document change the way that stress data is input to the core part of the in-house reshaping routines. Changes have also been made to the way the new shape is output. The prior code made use of a commercial package scripting language for performing these functions and the use of this language was the source of compatibility problems. These input and output functions have been rewritten using a generic programming language that is in widespread use across scientific computer hardware. A number of 2D and 3D example problems involving stress concentrations of various types have been optimised in order to demonstrate the successful operation of the converted code. These range in complexity from open-boundary fillets and closed-boundary holes, to a more complex case involving interacting holes. All of these test cases have been successfully solved, indicating that the converted shape optimisation code is operating correctly.

The Defence outcome of this work is that the DST Group shape optimisation capability can be maintained well into the future for further application to Australian Defence Force (ADF) aircraft. These implementations will help manage ageing aircraft structures with attendant benefits of reduced cost and increased availability. The technology is also well suited to the design of new aircraft and this research done by DST Group makes a significant contribution to the body of knowledge that may be used by designers of new aircraft.

Authors

Robert Kaye
Aerospace Division

Mr Robert Kaye joined what was then the Structures Division of the Aeronautical Research Laboratory in 1990 as a structural engineer with a background in full-scale testing. The first three years at DSTO were spent in evaluation of bonded repairs primarily using finite element methods. Included in that was the analysis of repairs to fuselage skin lap-joints, wing skin planks and bulkhead frames. More recently, he has been involved with structural and mechanical aspects of full-scale fatigue test installations. In particular, he played a key role in the development of a low-stiffness air-spring for the application of static load to a vibrating airframe. This work was followed by a period of several years doing research and development into the alleviation of stress concentrations by way of adaptive shape optimisation. This has been applied to concave metallic free boundaries and to the adhesive layer and end tapering of boron patches bonded to metallic structure. Upon his retirement, he is now a DST Honorary Fellow in Aerospace Division.

Witold Waldman
Aerospace Division

Mr Witold Waldman completed a BEng (with distinction) in Aeronautical Engineering at the Royal Melbourne Institute of Technology in 1981. He commenced work in Structures Division in 1982 at what was then the Aeronautical Research Laboratory. He has published a number of papers and reports, and his experience has focussed on stress analysis using finite element and boundary element methods, structural mechanics, fracture mechanics, computational unsteady aerodynamics, structural dynamics testing, digital filtering of flight test data, nonlinear optimisation, and spectral analysis. His recent work has been in the areas of structural shape optimisation and the computation of stress intensity factors. He is currently a Senior Research Engineer in the Structural and Damage Mechanics Group in the Airframe Technology and Safety Branch of Aerospace Division within the Defence Science and Technology Group, Department of Defence.

Contents

This page is intentionally blank

1. Introduction

Since 1997, research has been conducted within the Aerospace Division of DST Group relating to the use of adaptive shape optimisation to minimise stress concentrations in aircraft structures. At the core of the method is the removal of material in the vicinity of a stress concentration to produce a region of boundary having minimised and constant hoop stress. All forms of the method have involved an iterative procedure consisting of a finite element solution and some code to change the boundary shape based on the boundary hoop stresses. General details of the method and some of the early implementations, including repairs to the wing pivot fittings of F-111 aircraft that were previously in service with the RAAF, are described by Heller et al. [1] and McDonald and Heller [2]. A more recent successful application was the design of a rework repair to the F/A-18 LAU-7 missile launcher guide rail, described by Heller et al. [3].

The original version of the shape optimisation method used the PAFEC finite element solver and was a basic single stress peak method [4]. This minimal version was upgraded with the inclusion of a capability to solve problems that involve multiple stress peaks, but still using the PAFEC solver, as described in more detail in [5], [6] and [7]. Further developments included the treatment of multiple load cases and the addition of a capability for enforcing a minimum radius of curvature constraint on the optimised geometry. A Patran/Nastran version of the method that includes these developments is described by Braemar [8], and it has been successfully used for several years to solve a number of practical problems, such as the one reported by Heller et al. [3].

Recently, a limitation of the Patran/Nastran version of the method has become more significant and has led to the requirement for the work that is described in this document. The Patran part of the Patran/Nastran method uses scripting code, which is written in Patran Command Language (PCL), to extract the stresses from the Nastran solution and to build the modified Nastran input deck for the next iteration, including the recreated finite element mesh [8]. This PCL code was written for the 2003 version of Patran, and it was found to be incompatible with later versions of Patran. The present report describes the replacement of the PCL part of the shape optimisation code with a number of FORTRAN routines. The part of the code that calculates the new boundary nodal positions was written in the C programming language, and it has been retained with minimal modifications. At this stage, the new FORTRAN program code interfaces with the Abaqus finite element solver, but it is entirely amenable to modification for use with Nastran as well.

Apart from not being dependent on the version of Patran that is in use, this FORTRAN/Abaqus implementation does not use Patran at all, so a Patran licence is not required during run-time. The new FORTRAN source code, as well as the pre-existing C source code, is expected to be highly portable across Linux platforms, and also to Windows platforms should the need arise.

Described in this document is the development of the code to the point where a number of successful solutions to closed-boundary and open-boundary problems have been achieved. Three examples of closed-boundary problems are presented, as well as two examples of open-boundary problems. These examples included multiple stress peak and multiple load case (robustness) type problems, as well as 2D and 2.5D type problems.

2. Comparison of the Patran/Nastran implementation with the FORTRAN/Abaqus implementation

The prior Patran/Nastran implementation is summarised in the flow chart shown in Figure 1. The entire process sits within a PCL program with the C program that calculates new boundary node positions being called from within that PCL program [8]. The use of PCL allowed for easy access to the Patran meshing functions, and so it was convenient to recreate the mesh and the geometry at each iteration. This provided for a high level of control of various mesh characteristics, such as the mesh density and the variation in density with distance from the boundary. By generating a new mesh at each iteration the possibility of mesh distortion was much reduced. The use of PCL also enabled the use of the Patran report writing functions for convenient output of the stress results to a text file.

A primary difference of the Patran/Nastran implementation with the FORTRAN/Abaqus implementation introduced here is that the movable part of the mesh is retained between iterations, with just the nodal coordinates being edited by small amounts to allow for the shape change. The characteristics of the updated mesh are therefore determined by the initial mesh and that pattern is retained throughout the optimisation process. This approach, which is shown in Figure 2, is more direct and provides some savings in execution time.

Another difference with the proposed implementation is that the stresses from the finite element solution are extracted directly from the Abaqus output files using a FORTRAN user subroutine that runs in conjunction with the Abaqus solver.

Neither method has the capacity to deal with mid-side nodes, so the movable part of the mesh is limited to 4-noded 2D elements or 8-noded 3D elements.

3. Initial steps to set up 2D problems

The finite element model should be constructed so that all the movable nodes lie in the x-y plane. The location of the origin is not important. The movable boundary nodes should have matching stationary nodes (partner nodes) at the edge of the region to be remeshed (movable mesh region). Intermediate nodes should lie on straight lines between their nearest boundary node and partner node, as shown in Figure 3. After the finite element model is solved to obtain the stress distribution, new nodal coordinates will be calculated for the type 1 boundary nodes. Again, the intermediate type 3 nodes will be positioned on a straight line between the newly located boundary node and its matching partner node (type 2). These intermediate nodes retain their relative position along the straight line as determined from the initial mesh, i.e. the ratio of the distance along the line to the line length is calculated from the initial mesh and stored for use in positioning the intermediate nodes.

The initial Abaqus input deck needs to be created in Patran with the name `patstart.inp`. Multiple load cases can be included if necessary and Abaqus output should be directed to the `.fil` file in binary format. User defined output requests are not required, as all output requests are later overwritten with a single request for computing principal stresses at nodes in the Abaqus input file.

The type 1 boundary nodes and their matching type 2 partner nodes need to be defined in two text files, one called `nodes.ini` and the other called `pnodes.dat`. The format for these two files is identical. The number of nodes is given on the first line, followed by lines containing the node id and the *xyz* coordinates of that node, one node per line, with the nodes provided in anticlockwise path order.

The easiest way to obtain the *xyz* coordinates of the nodes is to use the *`show/node/location`* function in Patran. Assuming that the region has been automeshed, the node numbers will be sequential within each automeshed surface, and a sequential node list can be entered using the standard Patran shorthand format, for example:

```
node 251:242:-1, 464:473, 585:593, 231:221:-1
```

Running the executable file `addz` will put zero values in for *z* nodal coordinates where they have been omitted in the Abaqus input deck. This routine reads from `patstart.inp` and writes the new version of the file to `allzstart.inp`.

Running the executable file `binid` will write the file `start.inp` while reading from the file `allzstart.inp`. All nodes will have a comment line added after the node data line that includes the node type as an integer, the node pair associated with the node (also an integer) and a real value called `posnode` for type 3 nodes. The value of `posnode` indicates the distance of the type 3 node along the line between its associated boundary node and partner node expressed as a proportion of the total distance outwards from the boundary node. This routine counts the type 3 nodes that have been found so that the user can check that the number is correct. A tolerance value `type3tol` can be edited in the source code `binid.f` if the number is incorrect. One option to recompile is `./compall.sh`. The program `binid` will also remove all output requests written into the file by Patran and put in a single request for computation of the principal stresses at the nodes.

4. Solution procedure for 2D problems

Documentation for the existing Patran/Nastran implementation is provided in [8] and is mostly still relevant to this implementation. The part of the present process that calculates the new positions of the boundary nodes at each iteration is nearly the same as the C code from `optim_func_v41.c` [8]. No changes have been made that affect the calculation of the new nodal coordinates. The changes are related to integrating the C code with the new FORTRAN routines as opposed to the old PCL routines. The new version is named `optim4.c` for the source code and `optim4` for the executable equivalent.

The solution procedure for this proposed implementation is controlled by a simple Linux shell script called `opscript.sh`, which is summarised below and is listed in full in Appendix A.

Summarised listing of opscript.sh

```
#!/bin/sh

cp nodes.ini nodes.dat
cp start.inp opjob.inp
```

```
rm -f convergence.dat
rm -f znodes???.dat

maxitns=200

for (( i = 1; i <= maxitns; i++ ))
do
  mv opjob.inp temp.inp
  rm -f opjob.*
  mv temp.inp opjob.inp

  abaqus job=opjob user=getsigq_ww cpus=1 interactive

  ./fsig    # Formats and collates stress???.dat files
  ./rednf   # Reduces format of nodes.dat and puts in nodes.opt
  ./optim4  # No mods have been done that affect function
  ./expnf   # Expands format of nodes.opt and puts into nodes.dat

  cp opjob.inp opjob.temp
  ./bdeckq  # Builds new Abaqus input deck using new nodes.dat
  rm -f opjob.temp

  ./wrconv  # Writes peak hoop stress. Adds one line each iteration
  ./wrshape # Stores nodes.dat for each iteration in znodes???.dat
done
```

The first line of this script identifies the file as being a shell script to Linux. The next two lines copy initialisation files `nodes.ini` and `start.inp` to their run-time versions `nodes.dat` and `opjob.inp`. The file `nodes.dat` varies with each iteration and contains the updated positions of the movable boundary nodes. The file `opjob.inp` also varies with each iteration with new nodal positions based on the file `nodes.dat`. The next two lines remove files from previous runs.

The first three lines inside the `for` loop remove Abaqus files from the previous iteration so that Abaqus does not stop the script and ask the user whether to remove these files. The next line is the Abaqus command line. The `interactive` option directs Abaqus to run in the foreground and prevents execution from proceeding to the next step in the shell script before the Abaqus run has finished. The `user=getsig` option compiles and runs a user defined subroutine `getsig.f` in conjunction with the Abaqus job. This routine extracts and averages the nodal principal stresses for the list of movable boundary nodes contained in `nodes.dat`. This routine runs at the end of each load case and puts the stresses in files (one file per load case) having names of the form `stress??.txt`, i.e. `stress01.txt`, `stress02.txt`, etc. The next line in the shell script is the `fsig` program which collates the above stress files into a single stress file called `stress.rpt`. This file is also formatted by `fsig` to be compatible with the `optim4` shape change program.

The program `rednf` removes the first line and the first column from `nodes.dat`. It also adds a column after the x, y and z coordinates that can be used for constraining the movement of individual nodes. These constraints are not normally used, and this column is filled with zeros at this stage. `rednf` writes the reduced version of `nodes.dat` to the file `nodes.opt` which is read by `optim4`.

One key input to `optim4` is the file `parameters.dat`, which contains a number of option settings. Many are not used by `optim4.c` as they relate to the use of the Patran PCL code that has now been replaced. Details relating to `parameters.dat` are given in Appendix B. With the exception of editing the path of the current directory, it is recommended that this file be left unchanged in the first instance.

The program `expnf` reformats the movable boundary nodes from `nodes.opt` back into the original format and writes to `nodes.dat` noting that the x and y coordinates will have been modified slightly by `optim4`. It should also be noted that the format of `nodes.opt` is modified by `optim4.c` as well as the nodal positions. The inclusion of `rednf` and `expnf` either side of `optim4` in the shell script ensures that the format of `nodes.dat` is consistent throughout the run.

The program `bdeck` reads the current Abaqus input deck from `opjob.temp` and writes a new version to `opjob.inp` containing the updated boundary shape from `nodes.dat`. The intermediate mesh is also updated based on the mesh data comment lines in the Abaqus input deck `start.inp`, i.e. the type 3 nodes are relocated on the line between their associated pair of boundary node (new location) and partner node at a position along the line based on the value of `posnode` (also from the mesh data comment line). Having the mesh data comment lines in the Abaqus input deck greatly reduces the amount and complexity of the source code of `bdeck`.

The programs `wrconv` and `wrshape` create a record of the job for use during run time and after completion. `wrconv` writes out the peak tensile hoop stress as an appended line to the file `convergence.dat` at each iteration. Viewing this file during run time is useful for assessing whether the job is progressing as expected and consequently deciding whether to terminate the job prematurely if necessary. `wrshape` stores the current version of `nodes.dat` in files with names `znodes???.dat`, i.e. `znodes001.dat`, `znodes002.dat`, etc. Any of these files can be used as the input file for a stand-alone run of `bdeck` to create an Abaqus input deck for the chosen intermediate shape of a completed job. To do this, it is of course necessary to edit the file names in `bdeck.f` and recompile.

Some further detail with regard to the instructions is provided in Appendix C.

5. Solution procedure for 3D problems

Although some solutions are applied to 3D models, the computed shape change is always 2D in nature, as shown in Figure 4. In these cases, the peak stress through the thickness of the component is used to determine the shape change, and this results in slightly different shapes compared to the 2D solutions.

For solutions applied to 3D models, some additional steps are required so that the correct stress value is used as the basis for calculating the new nodal positions, and so that nodes having the same x and y positions move together in the x and y directions.

When setting up the 3D problem, the three programs `binid1q`, `binid2q` and `binid3q` are run in place of the `binid` program. The program `binid1q` finds and labels the type 1, 2 and 3 nodes, reading from `allzstart.inp` and writing to `type123.inp`. The program `binid2q` finds and labels the type 4, 5 and 6 nodes where the type 4 nodes have the same x and y

values as the type 1 nodes, and similarly for type 5 and type 2 nodes and type 6 and type 3 nodes. The program `binid3q` writes the boundary node ids in groups having the same x and y coordinates to a file called `bndall.nds`. This file is read by a modified version of the `getsig` routine, which is called `getsigq`, that finds the maximum stress through the thickness for each group of boundary nodes. A modified version of the program `bdeck`, called `bdeckq`, is used to edit the nodal positions of the 3D mesh. FORTRAN programs that have been modified for use with 3D problems have had the letter `q` added to their file names. Programs with names that do not end with a `q` are the same for both 2D and 3D problems.

Some further details relating to running 3D problems is provided in Appendix D. Source code listings of the various FORTRAN and C programs are provided in Appendix E.

6. Example problems

Five example problems have been solved with the purpose of demonstrating that the stress data is being correctly input to the C-language part of the code (`optim4`), and that at each iteration the new Abaqus input deck has the nodal positions edited as expected within the movable mesh region. As the original functionality of `optim4` remains unchanged, the numerous features and options that are fully internal to `optim4` are not entirely demonstrated by these example problems.

The various files that are associated with the solution of these example problems are located in Objective folder fAV1044425.

6.1 2D model of circular hole near stress concentration

In this section we proceed to optimise the shape of a small circular hole that is in the presence of a larger circular hole. This is a new problem that has not been analysed previously, and it was chosen in order to demonstrate the correct operation of the present implementation in solving closed-boundary problems. The problem definition is given in Figures 5 and 6 and it consists of a one-quarter model of a large square plate with a centrally-located circular hole and a smaller circular hole located nearby. The smaller hole is the subject shape that is to be optimised. Four-noded quadrilateral elements were used in the region of the small hole. Three-noded triangular elements were used elsewhere. The radius of curvature in the optimised smaller hole was constrained to be greater than ¼ of the hole radius. The smaller hole has two regions of tensile stress and two regions of compressive stress around its circumference. The shape changes generated by `optim4` concurrently minimise the peak stress in each of the four regions. It is common for holes in a 2D stress field to have these four regions.

The results of the shape optimisation are given in Figures 7–9. Figure 7 shows that the re-meshing that is applied between iterations is clearly working as intended, with all the intermediate (type 3) nodes maintaining their relative positions in the mesh. Figure 8 shows the stress contours of maximum principal stress for the optimal subject hole that is located near the circular stress concentration. Figure 9 compares major principal stress contour plots of the initial circular shape (left) and the optimal solution shape (right), plotted using the same stress contour scale. Figure 10 provides a comparison of the hoop stress around the boundary for the initial and the final optimal hole shapes. There has been an overall reduction in the peak stress of 21.54%. The results presented in Figures 9 and 10 indicate that

the stresses have been correctly provided to the C-language part of the code that performs the shape optimisation.

6.2 2D model of circular hole near stress concentration using multiple load cases

This problem is a repeat of the previous problem with five load cases applied, four of which serve to perturb the stress distribution around the hole by small angular amounts. This problem has been included to showcase that the proposed implementation is able to handle multiple load cases, and that it therefore can create robust solutions, i.e. the resulting shape provides the same stress reduction for local stresses that are orientated within a predefined tolerance.

A small rotation of the stress distribution in the region of the hole has been obtained by applying a small amount of shear stress in the region of the hole. In order to accomplish this, the nodal forces were applied around the hole region as shown in Figure 11. The addition of a shear stress whose magnitude was 1% of the applied longitudinal stress was found to rotate the local stress distribution by about 1° degree. The problem was set up with load cases with the stress distribution rotated anti-clockwise by 6° and 3°, and clockwise by 3° and 6°. The load case with zero stress rotation was also included, making up a total of five load cases. Plots of stress contours for two of the load cases are shown in Figure 12.

The final hole shape that was obtained after 200 iterations gave a stress reduction of 16.6%, and is shown in Figure 13. As expected, it has a slightly more rounded shape than the result for the single load case solution. The boundary hoop stresses for all load cases are plotted in Figure 14, and they show that this implementation of the method is clearly working as expected for this robust, multiple load case problem.

6.3 3D model of circular hole using multiple load cases

This example is similar to demonstrator problem 2, except that the process has been performed on a 3D model as shown in Figures 15, 16 and 17. Once again, five load cases have been used with varying amounts of applied shear stress, as depicted in Figure 15 and using the values given in Table 1. The hoop stresses obtained for the final shape are plotted in Figure 18. A larger version of the subroutine `getsig`, called `getsigq`, has been used, which finds the largest of the principal stress values through the thickness of the plate on the hole boundary. As described earlier in Section 3, before starting the iteration loop, programs `binid1q`, `binid2q` and `binid3q` have been run mainly for the purpose of labelling the nodes contained in the Abaqus input deck `start.inp`. The labelling of the nodes allows the program `bdeckq` to move through thickness groups of nodes in tandem, i.e. nodes with the same x and y coordinates are moved by the same amount.

6.4 2D model of axially-loaded notched fatigue test coupon

In order to demonstrate the use of this FORTRAN/Abaqus implementation to solve an open-boundary problem, the arrangement shown in Figure 19 was modelled. It consists of a one-sided axial load test coupon comprised of a rectangular plate with a notch consisting of a shallow radius scalloped out of the left-hand side. The length of the notch is L_n and its depth is $L_n/20$. The total length of the coupon is $5L_n/3$ and its width is $L_n/3$. A uniform axial load is applied at each end. The plots of the initial and the final meshes that are shown in Figure 20 serve to demonstrate that the re-meshing code is working correctly. Figure 21 shows the

stress contours of the major principal stress that were obtained for the initial (left) and the final optimal (right) notch shapes. As would be expected for this initial notch shape, only a small amount of stress reduction was possible.

A comparison of the normalised boundary hoop stress for the initial notch shape and the final optimal notch shape is shown in Figure 22, where a 2.63% reduction in the peak stress has been achieved. Here the normalised distance along the notch boundary is defined by $\xi = (x+L_n/2)/L_n$. As is to be expected for an optimal shape, there is now an extensive region of uniform stress, comprising over 90% of the notch boundary.

Taken together, the results that have been presented in Figures 20, 21 and 22 all serve to indicate that the input data to the C-language part of the shape optimisation code is correct.

6.5 3D model of axially-loaded notched fatigue test coupon

An open boundary problem modelled in 3D has been solved in a similar manner to the previous example. Even though the final shape looks similar to the 2D version, the use of a 3D model does affect the shape slightly. Again, the level of stress reduction achieved is small due to the starting shape being near optimal. The general arrangement and x-y coordinate system are shown in Figure 23. The width of the coupon test article is one-quarter of its length. An initial centrally-located circular notch on both sides has a length of $L_n = 7L/16$ and a depth of $D_n = L/16$. The uniaxial loading has been applied as a uniform traction load that has been applied by a nodal force at each node that is in contact with the testing machine grip.

The finite element model is semi-symmetric in nature, and models the left-hand side of the coupon. Figure 24 shows a plan view of the initial finite element mesh (left picture) and the final finite element mesh corresponding to the optimised fillet boundary (right picture). The movable mesh region is as indicated. Figure 25 shows an isometric view of the mesh corresponding to the optimal solution shape (semi-symmetric half model). Figure 26 shows the distribution of the normalised major principal stress along surface of the notch (maximum value through the thickness) for the initial and the optimal notch shapes. Here the normalised distance along the notch boundary is defined by $\xi = (x+L_n/2)/L_n$. For the optimal shape, there is an extensive region of uniform stress along approximately 65% of the length of the notch boundary. The optimised shape produces a reduction in the peak stress of 6.65%. Taken together, these results serve to further confirm that the extraction of stresses and nodal movements is working correctly.

As reported elsewhere, in recent times the present code has also been successfully applied to the design of a novel constant-stress fatigue test coupon [9] using the 3D shape optimisation capability described in the present report.

7. Conclusion

The PCL part of the previous Patran/Nastran implementation of the DST Group shape optimisation code has been successfully replaced with some FORTRAN code and a Linux shell script. The new implementation is portable to any computer that possesses FORTRAN and C compilers and a shell scripting capability.

The results of the example problems that have been presented here have successfully demonstrated that the shape optimisation process can be implemented without the use of Patran. This outcome has reduced complexity and eliminated problems of the code being dependent on the Patran version that is in use.

Five example problems have been presented in detail in Section 6, which encompass both 2D and 3D test cases involving closed-boundary and open-boundary cases. The results that were obtained show that the present implementation gives solutions that are consistent with the prior implementation.

This new implementation is at stage where the first correct solutions have been obtained. As such, it would benefit from additional future development to make the implementation more user friendly, with fewer files, fewer routines and less preparation required for the setting up of problems that are to be shape optimised.

In the course of doing this work, the shape-change part of the code has been isolated and it can now be run in a stand-alone manner. This may be beneficial for users who need to use just this part of the code and manage the extraction of stresses and nodal movements in some other manner.

8. References

[1] M Heller, M Burchill, R Wescott, W Waldman, R Kaye, R Evans, M McDonald. *Airframe Life Extension by Optimised Shape Reworking – Overview of DSTO Developments*. 25th ICAF Symposium, Rotterdam, 27–29 May 2009.

[2] M McDonald, M Heller. *Robust shape optimization of notches for fatigue-life extension*. Structural and Multidisciplinary Optimization, Vol. 28, pp. 55–68, 2004.

[3] M Heller, J Calero, S Barter, RJ Wescott, J Choi. *Fatigue life extension program for LAU-7 missile launcher housings using rework shape optimisation*. DSTO Technical Report DSTO-TR-2662, February 2012.

[4] M Heller, R Kaye, LRF Rose. *A gradientless procedure for shape optimisation*. Journal of Strain Analysis and Engineering Design, Vol. 34, No. 5, pp. 323–336, 1999.

[5] W Waldman, M Heller. *Shape optimisation of two closely-spaced holes for fatigue life extension*. DSTO-RR-0253, DSTO, 2003.

[6] W Waldman, M Heller. *Shape optimisation of holes for multipeak stress minimisation*. Australian Journal of Mechanical Engineering, Vol. 3, No. 1, pp. 61–71, 2006.

[7] W Waldman, M Heller. *Shape optimisation of holes in loaded plates by minimisation of multiple stress peaks*. DSTO Research Report DSTO-RR-0412, April 2015.

[8] R Braemar. *Code enhancements for the PATRAN/NASTRAN structural optimisation*. DSTO Internal Minute to M Heller, 20 May 2005.

[9] W Waldman, R Kaye, X Yu. *A modified constant-stress coupon for enhanced natural crack start during fatigue testing*. DST Group Technical Report, 2016.

Table 1: Load cases applied to 3D hole example problem

Load case	Applied shear stress
6° clockwise	$+\sigma_{yy}/315$
3° clockwise	$+\sigma_{yy}/630$
0°	0
3° anti-clockwise	$-\sigma_{yy}/630$
6° anti-clockwise	$-\sigma_{yy}/315$

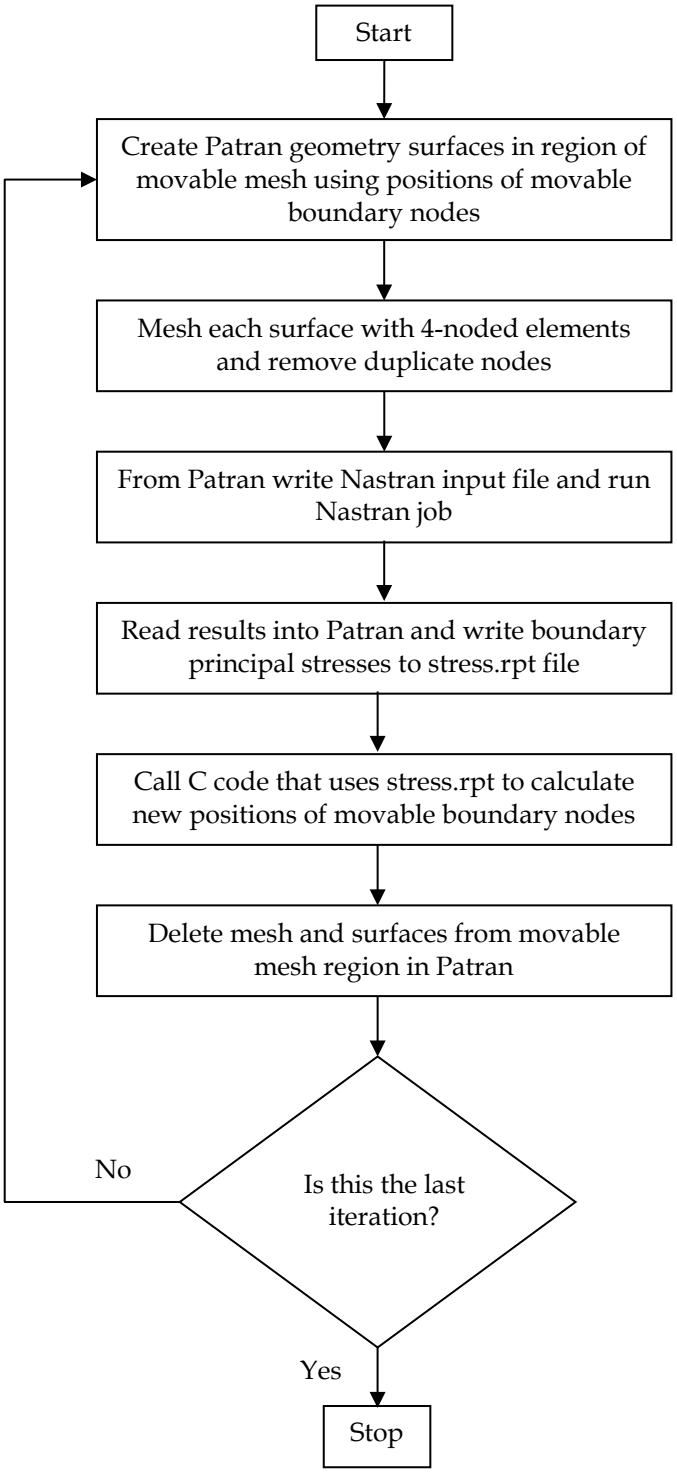


Figure 1: Flow chart describing prior Patran/Nastran implementation of the shape optimisation algorithm

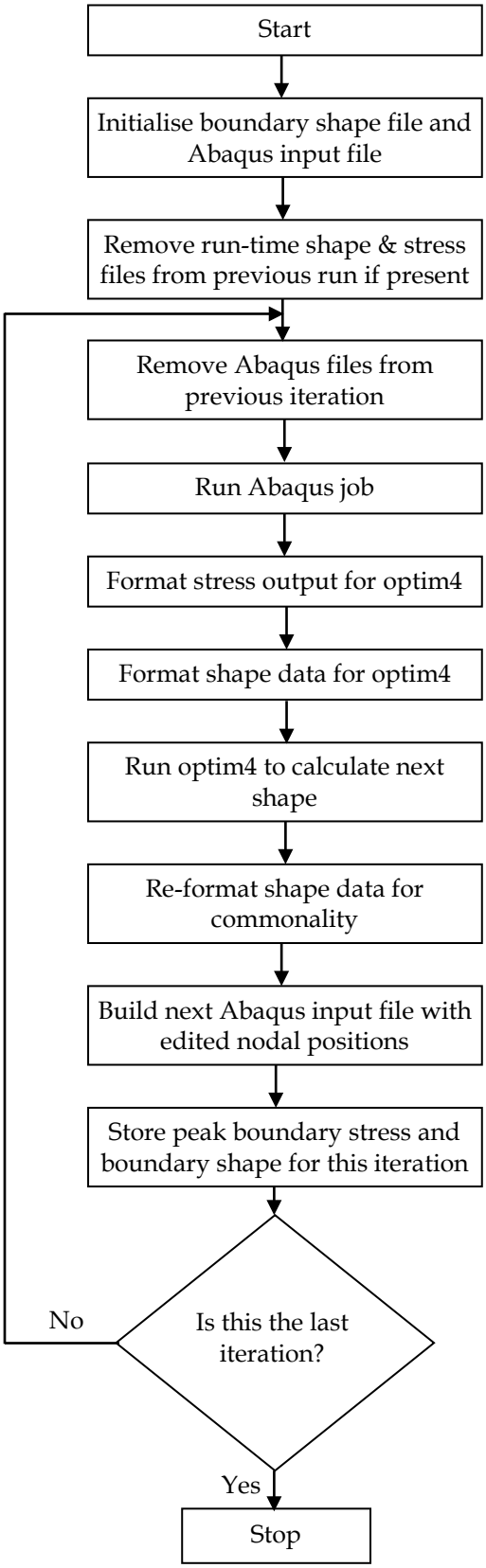


Figure 2: Flow chart describing the FORTRAN/Abaqus implementation of the shape optimisation algorithm

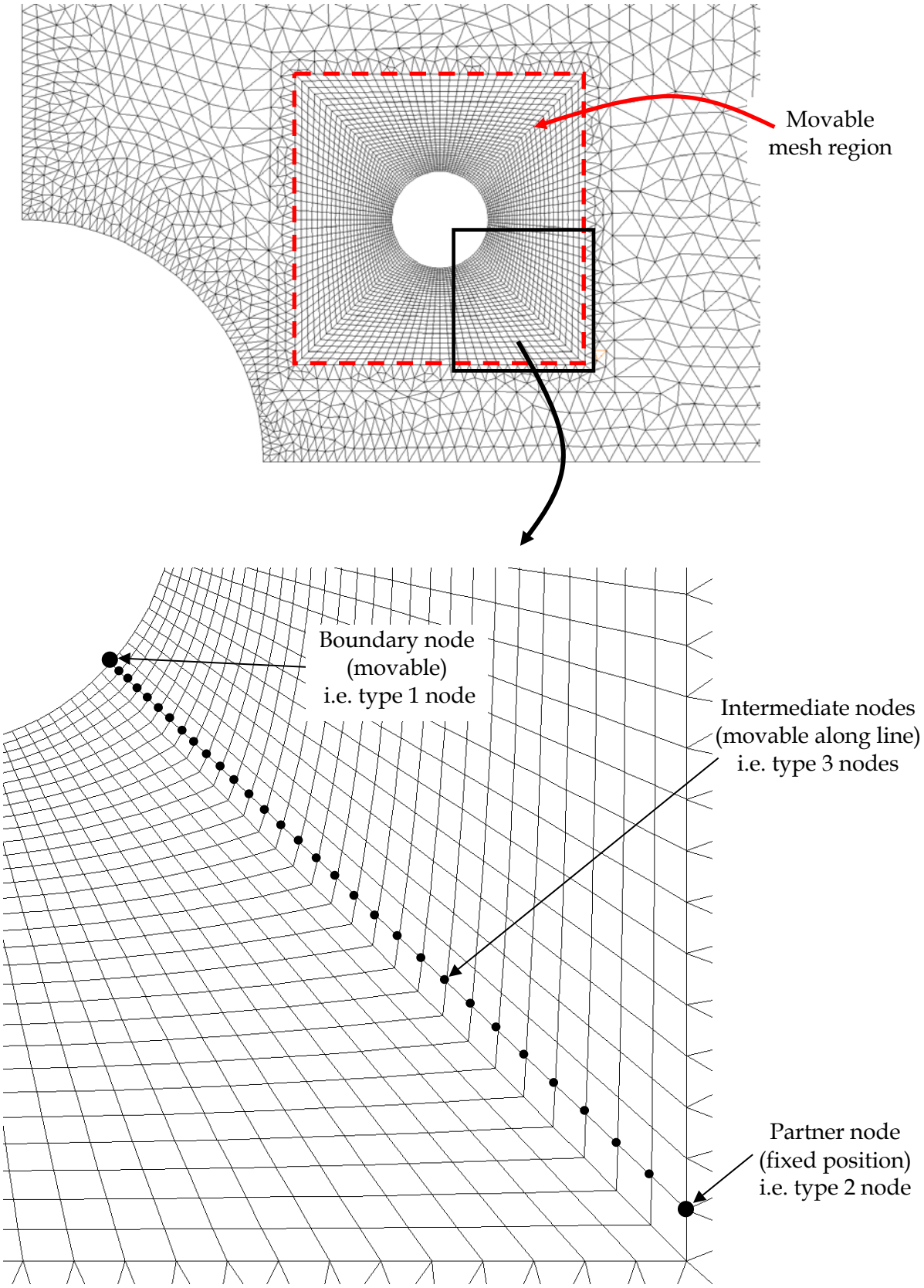


Figure 3: Some details relating to the movable mesh region

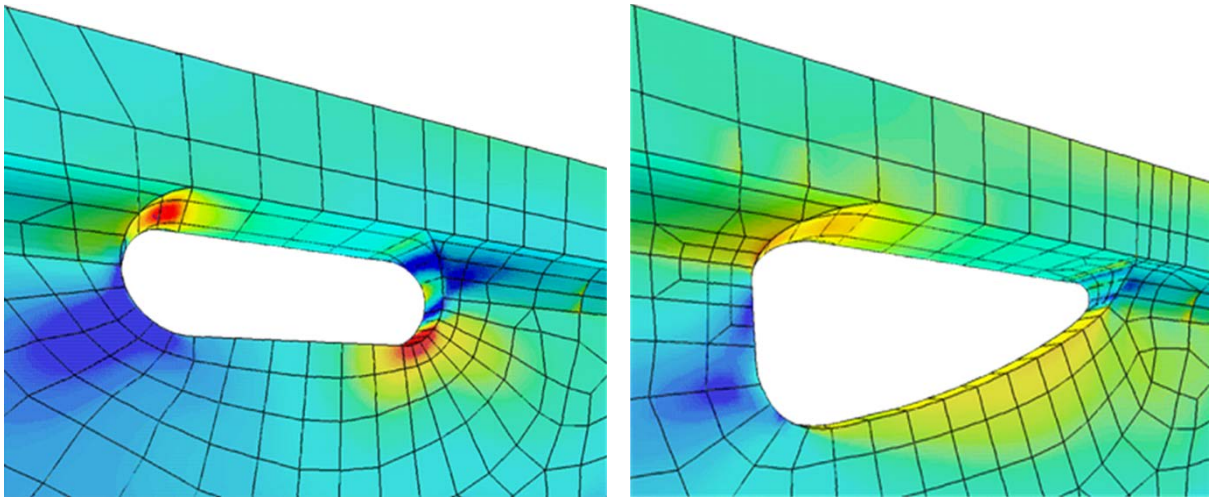


Figure 4: 2D shape optimisation applied to a 3D model

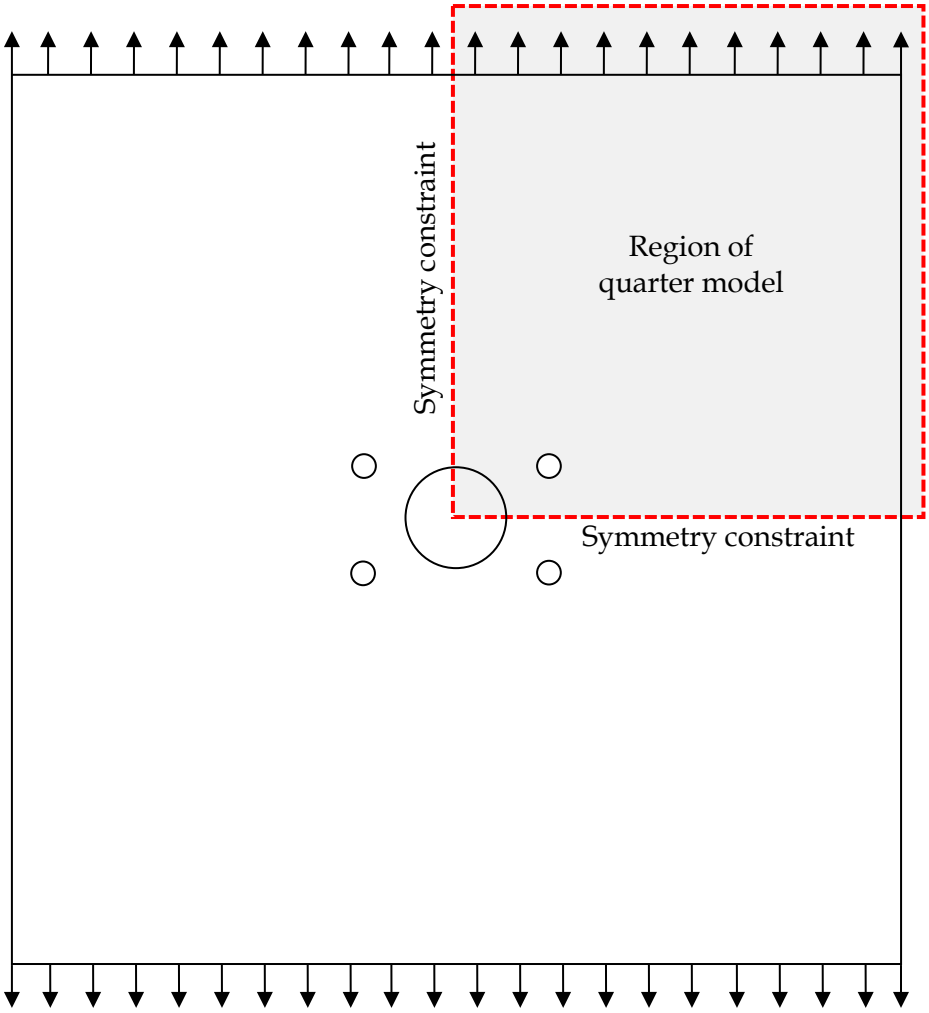


Figure 5: Geometry and loading arrangement for example problems 1 and 2 (not to scale)

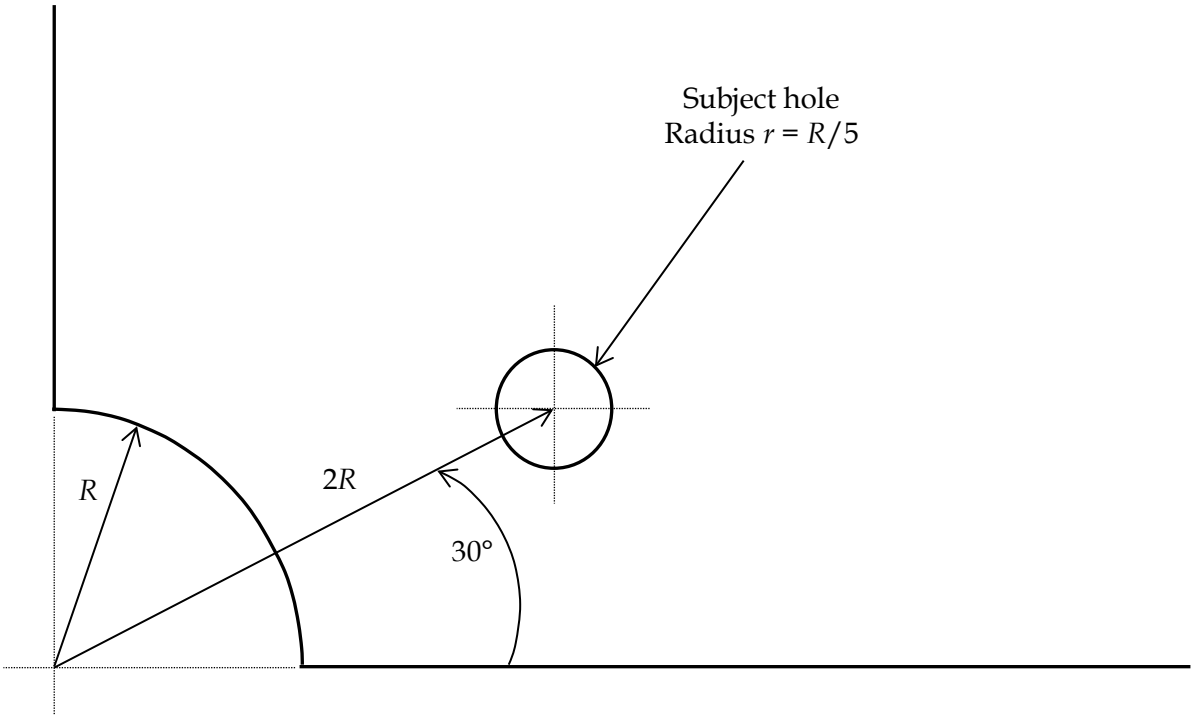


Figure 6: Geometric relationship between subject hole and large hole

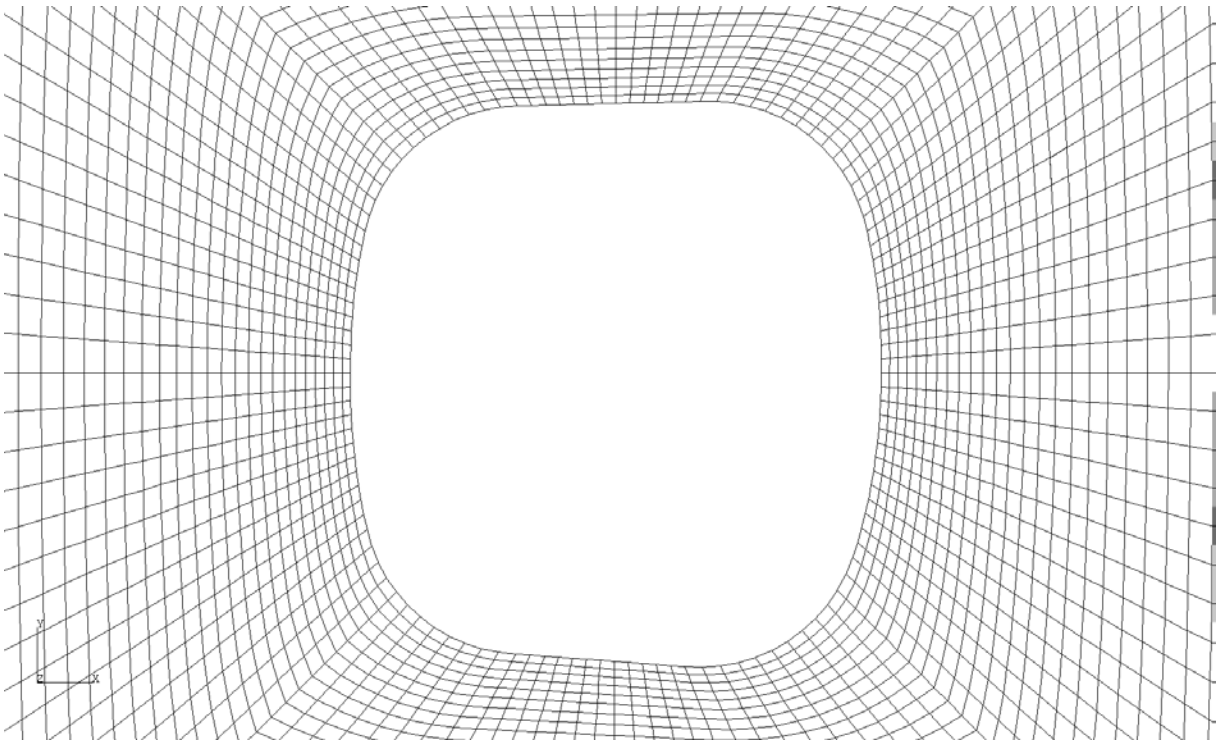


Figure 7: Solution shape for step size = 0.01r and iteration number = 200

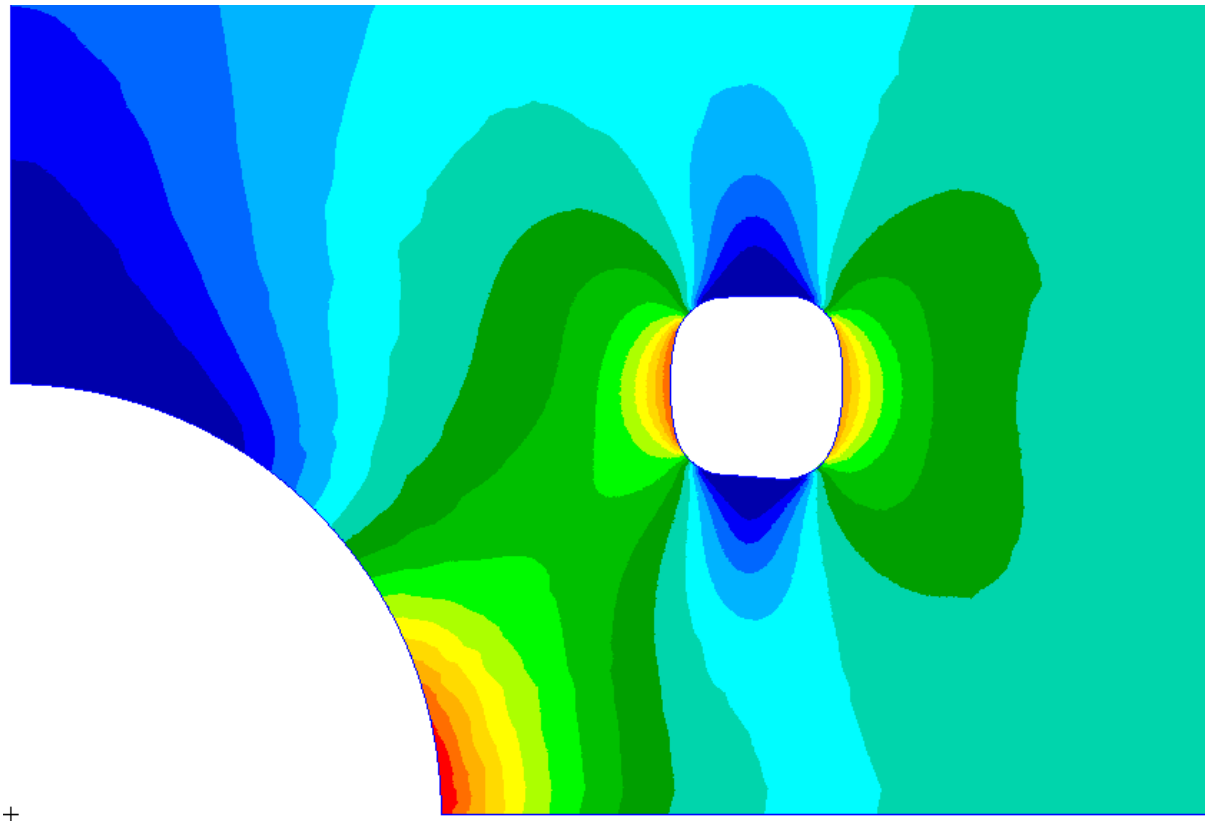


Figure 8: Stress contours of maximum principal stress for the optimised subject hole located near the circular stress concentration

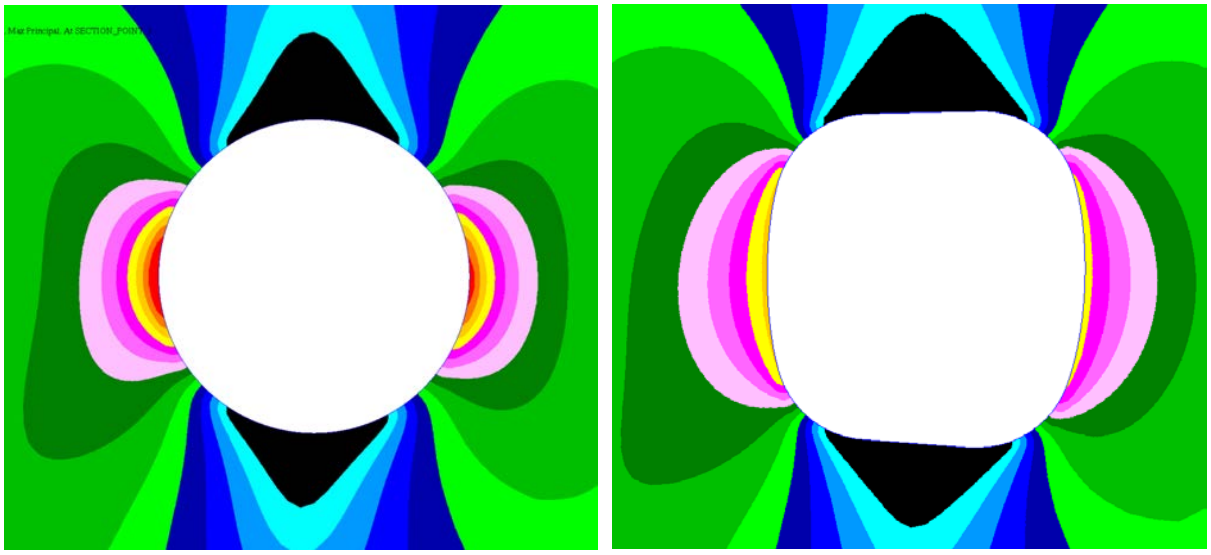


Figure 9: Comparison of major principal stress contour plots of the initial circular shape (left) and solution optimal shape (right), plotted using the same stress contour scale

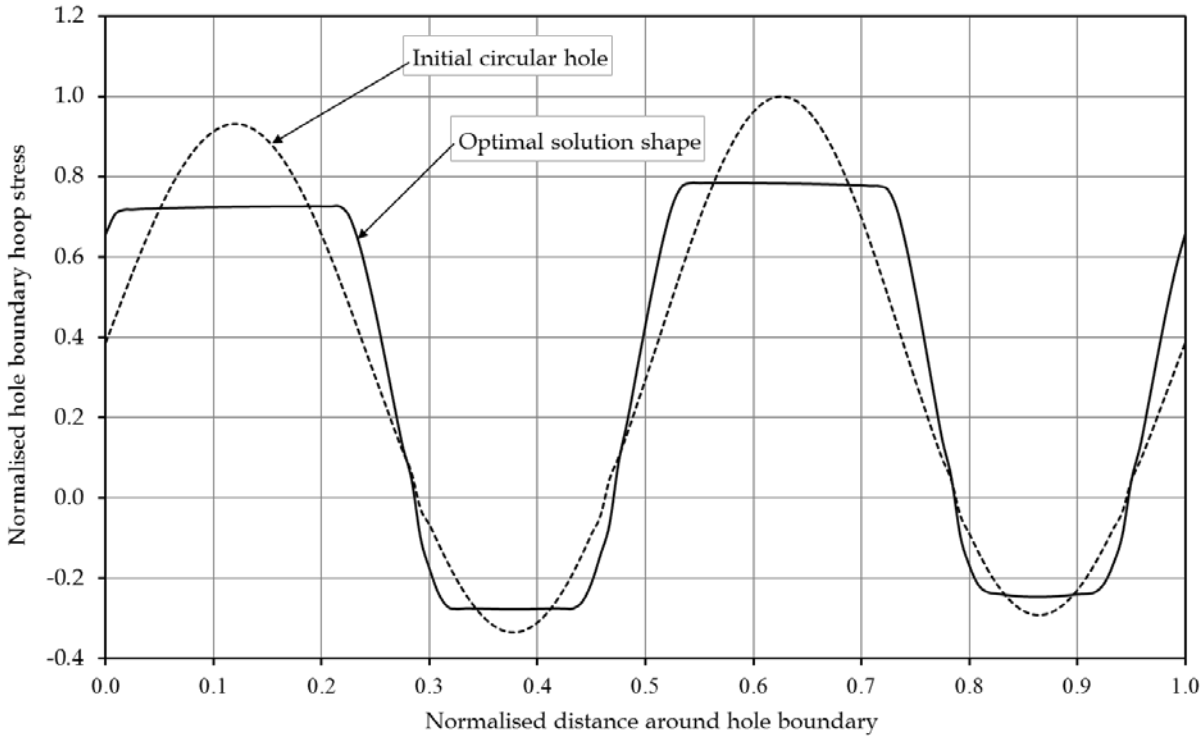


Figure 10: Comparison of boundary hoop stress for initial and final hole shapes showing the overall stress reduction of 21.54%

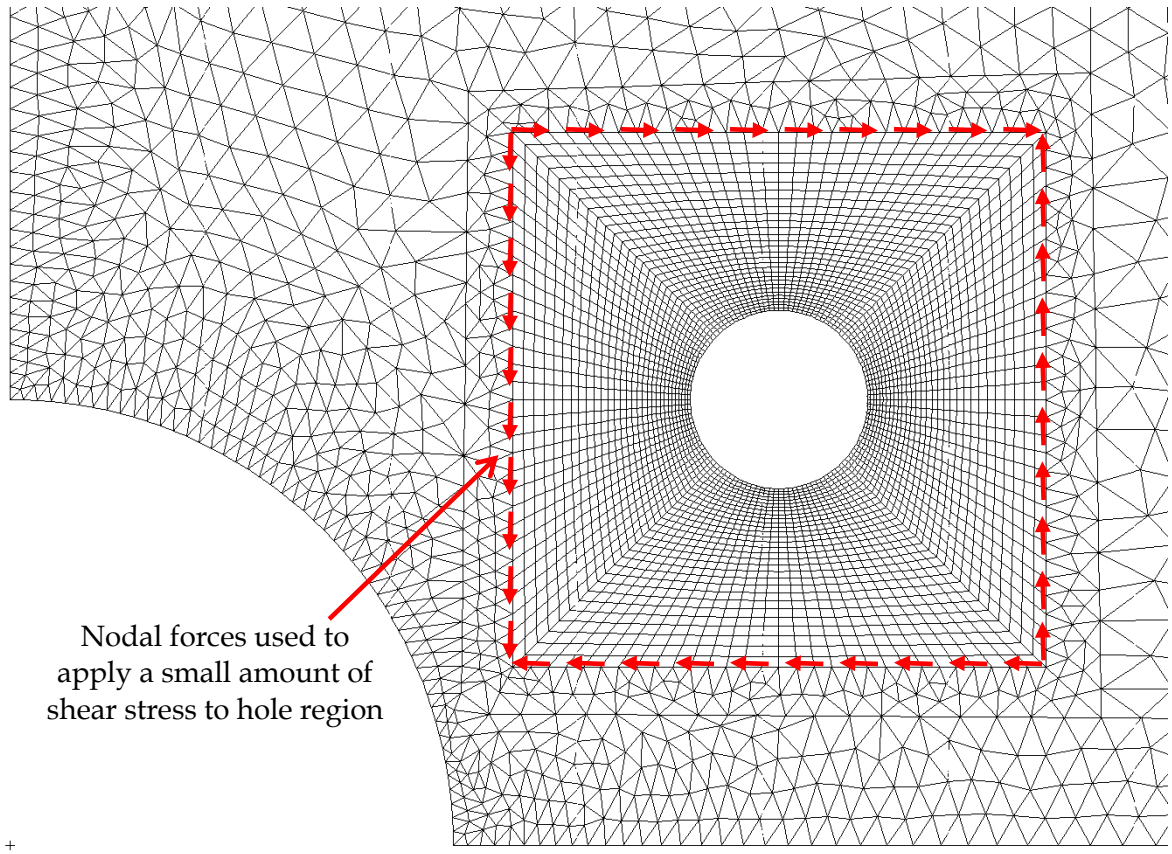
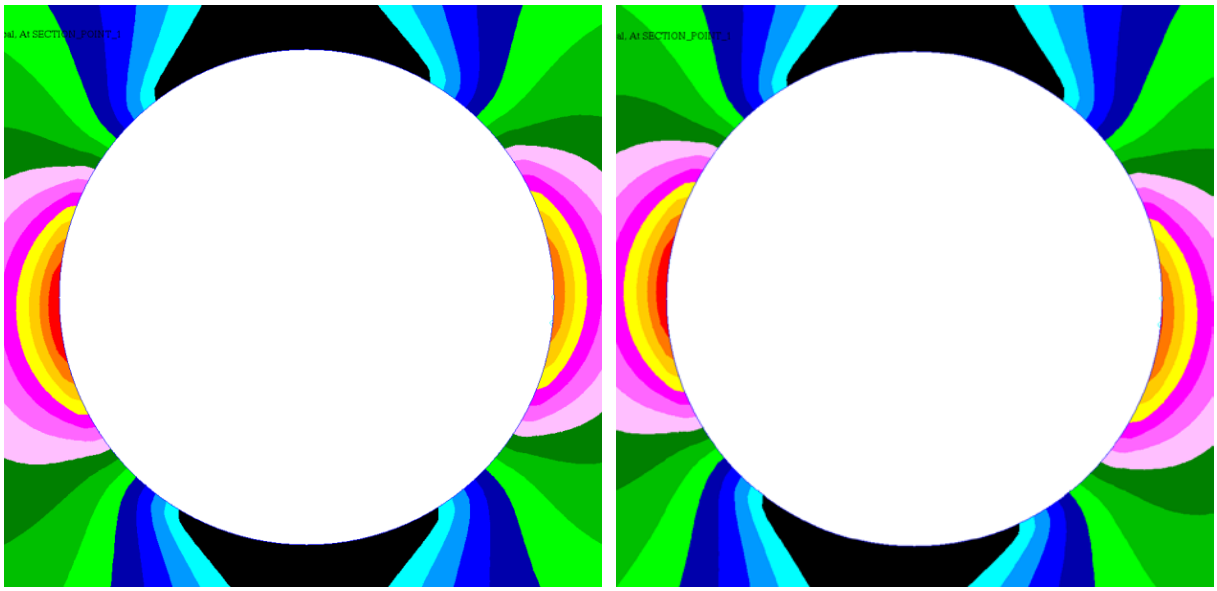


Figure 11: Arrangement of forces used to apply a small amount of shear stress to the region where the subject hole is being optimised



(a) (b)

Figure 12: Stress distributions rotated by (a) 6° anti-clockwise, and (b) 6° clockwise, plotted using the same stress contour scale

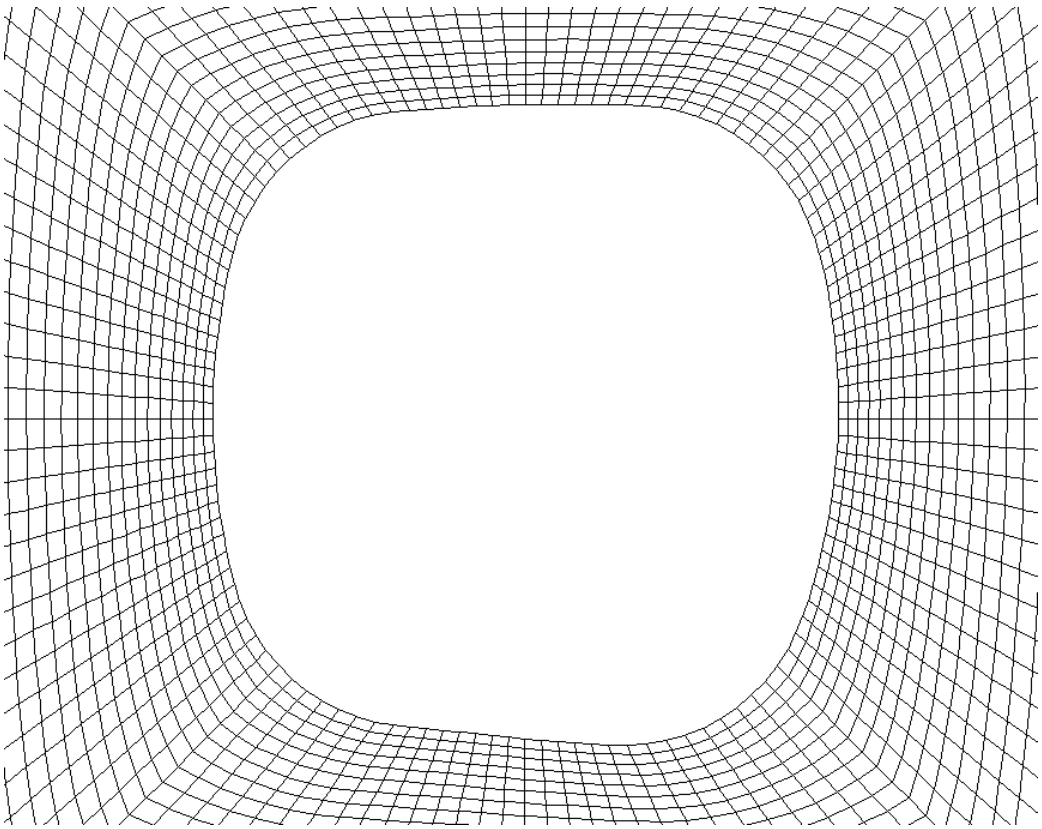


Figure 13: Solution shape at iteration 200 for hole near stress concentration with ±6° of robustness obtained using a step size of 0.01r

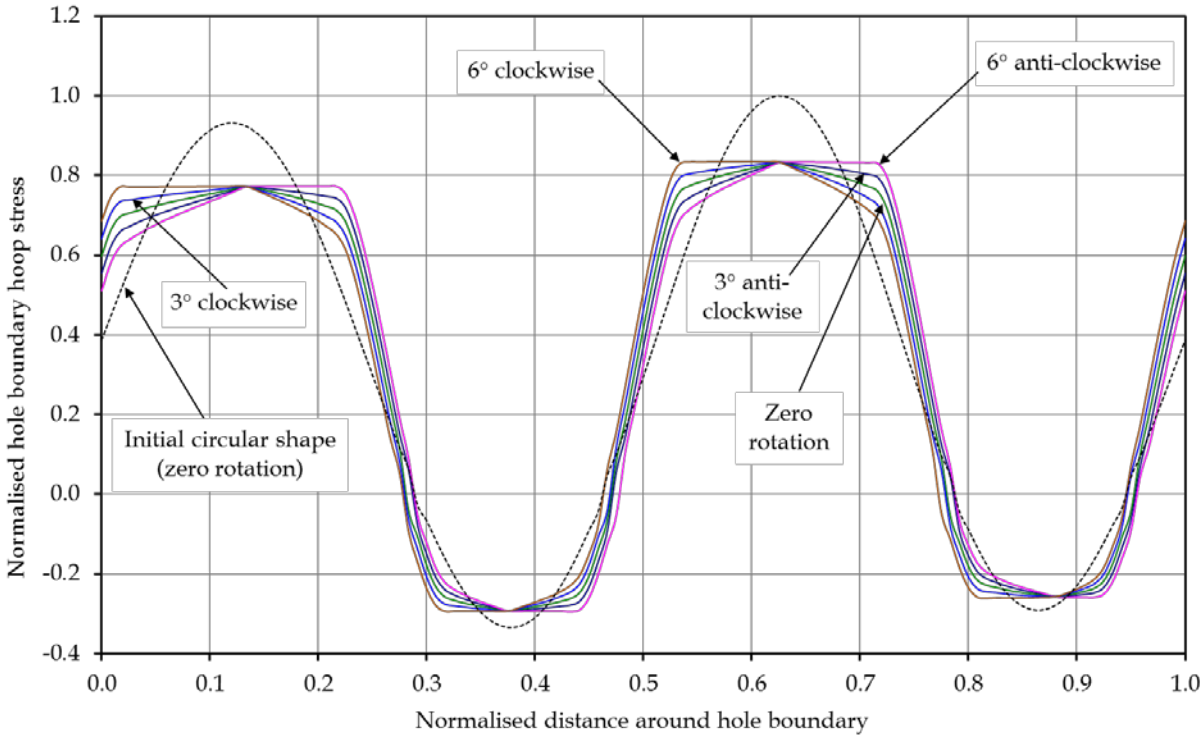


Figure 14: Comparison of boundary hoop stresses for initial and final hole shapes with multiple load cases showing stress reduction of 16.6%

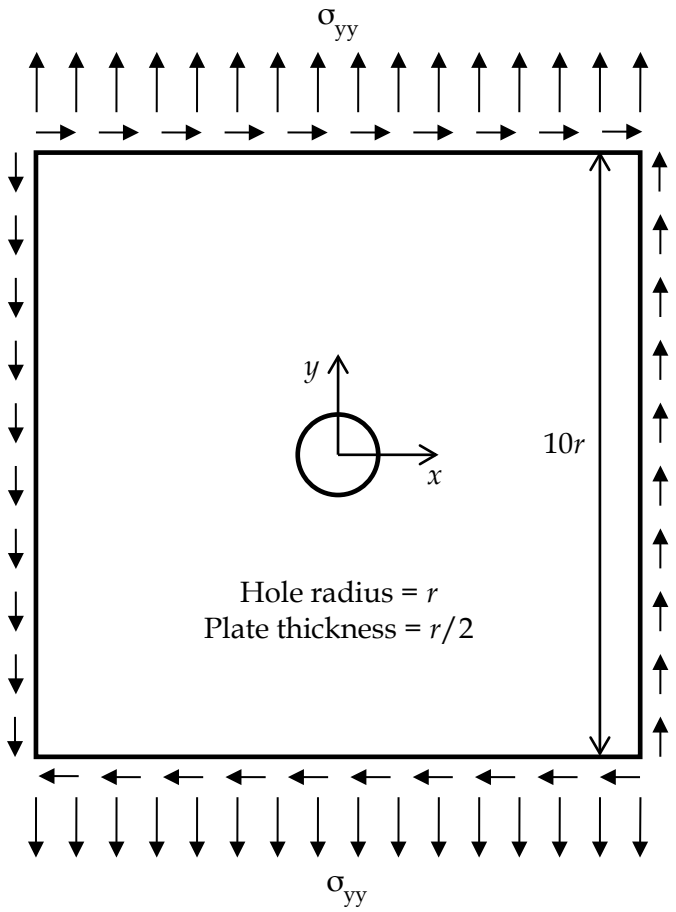


Figure 15: Geometry and loading arrangement

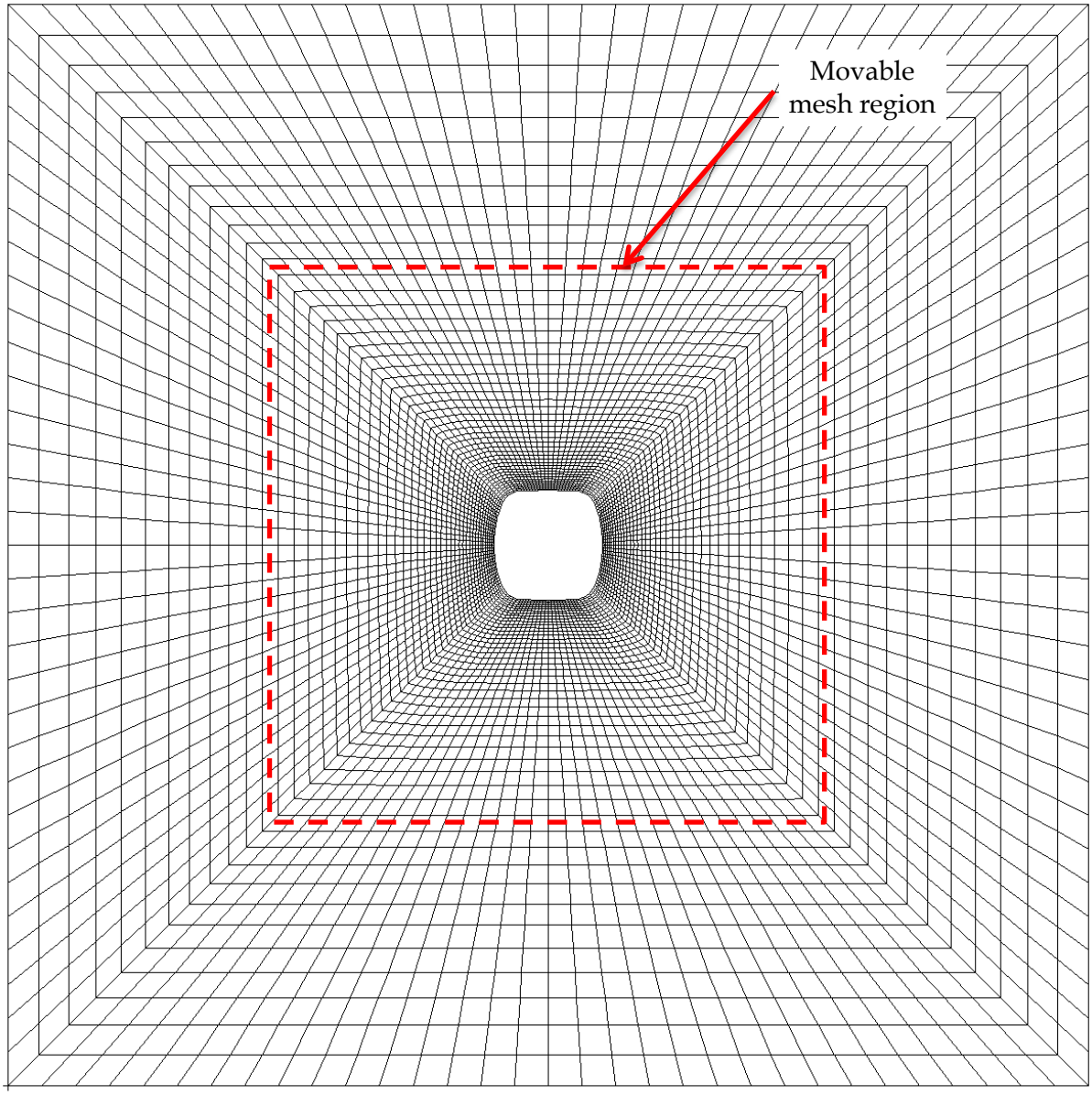


Figure 16: Plan view of the final shape of the finite element mesh

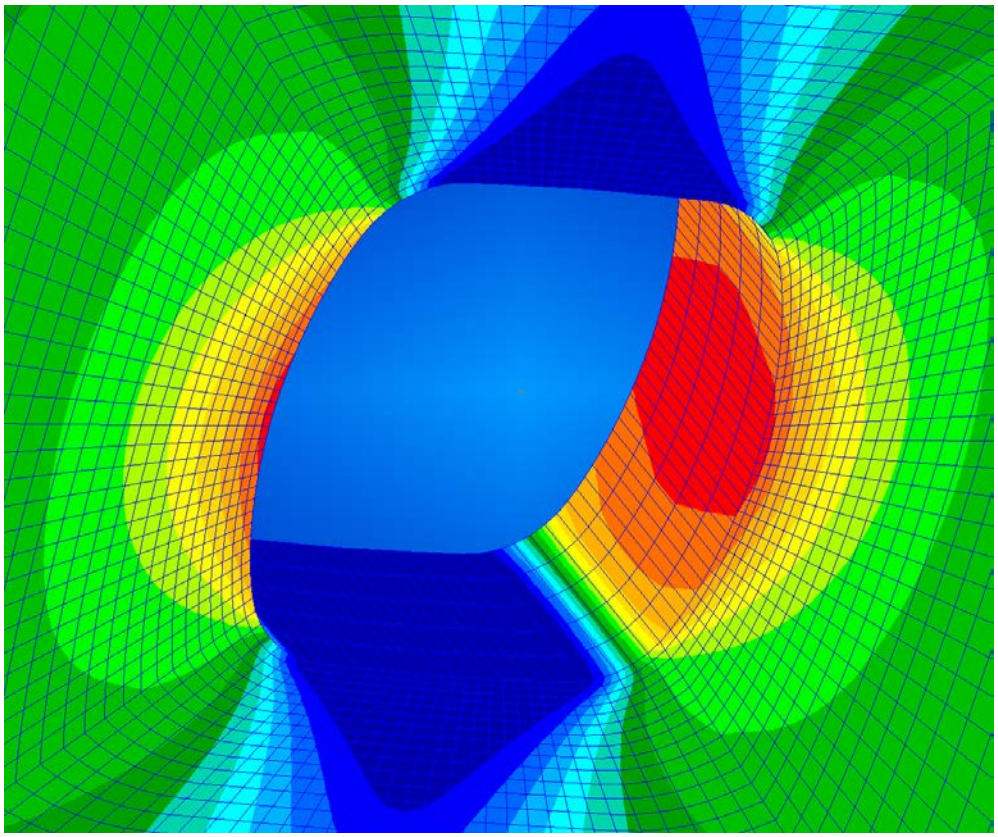


Figure 17: Isometric view of finite element mesh and contours of the major principal stress in hole region for the 0° load case (for final shape)

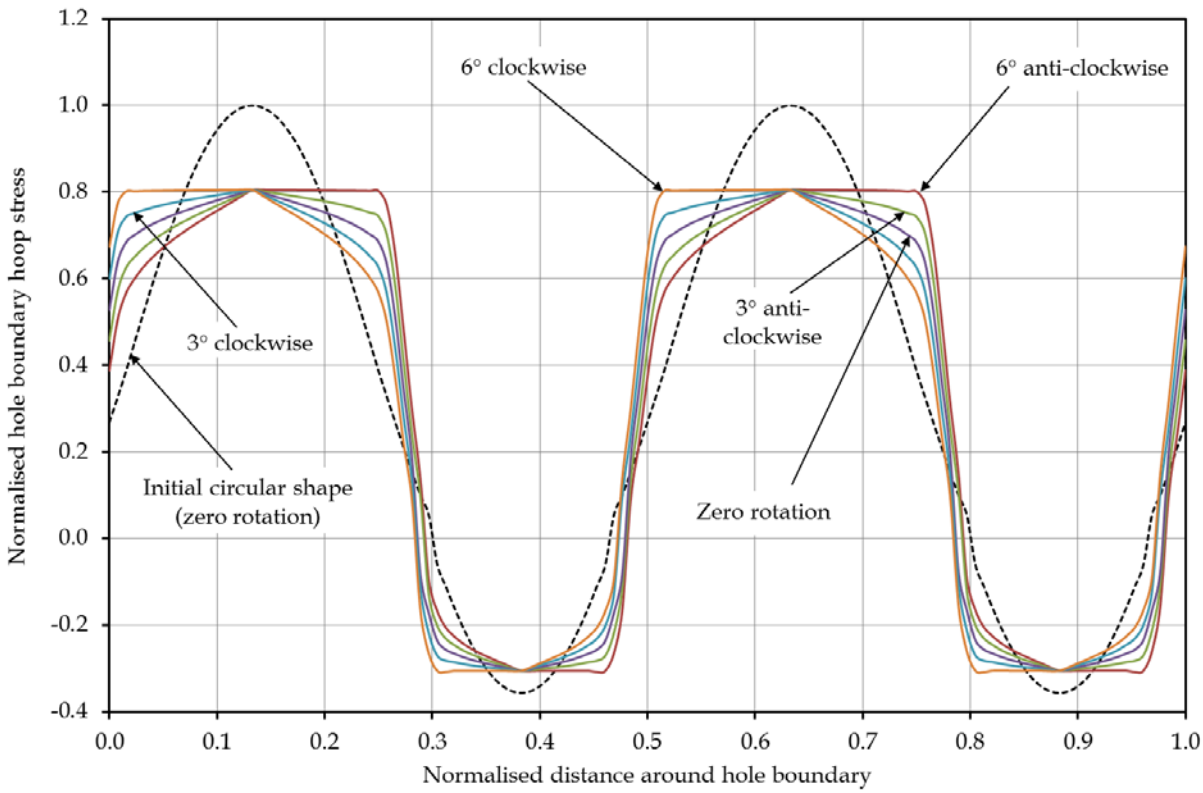


Figure 18: Comparison of boundary hoop stresses for the initial hole shape and the final hole shape with multiple load cases

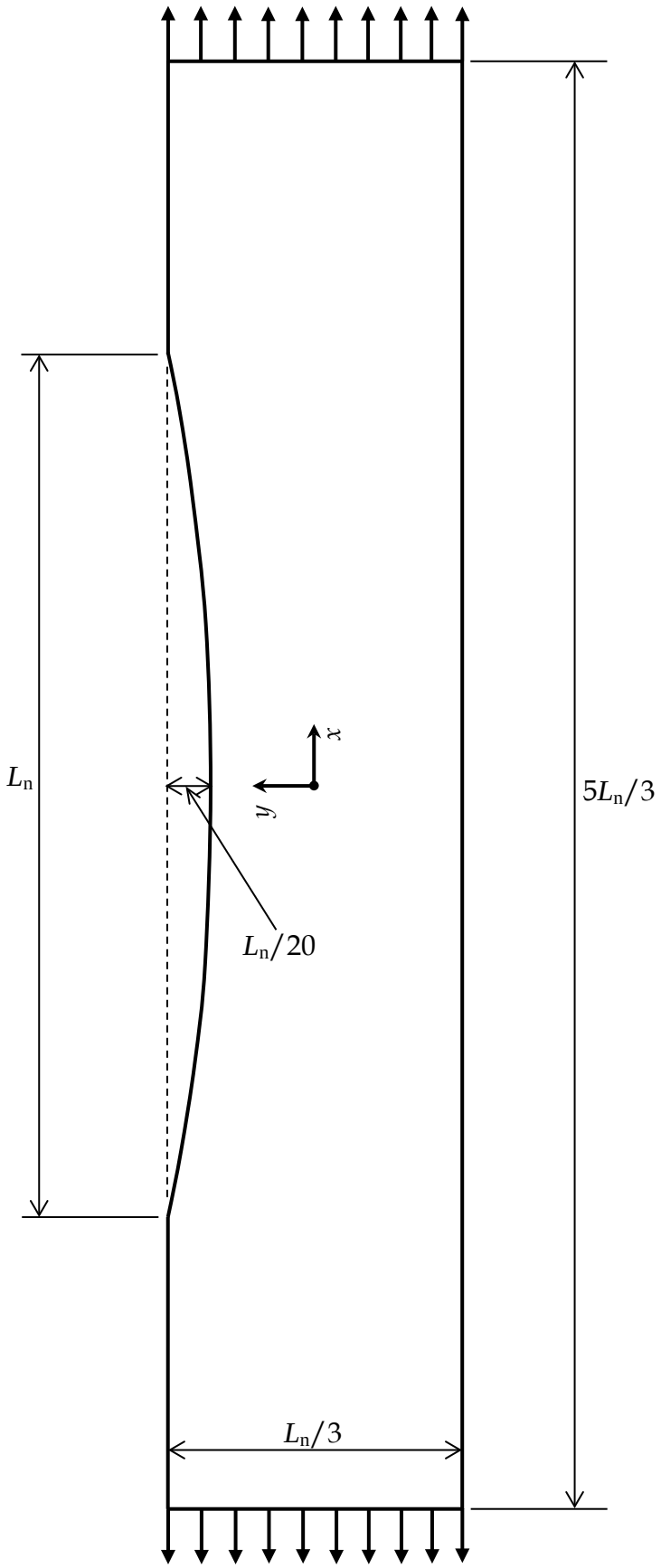


Figure 19: Loading and general arrangement for open-boundary problem

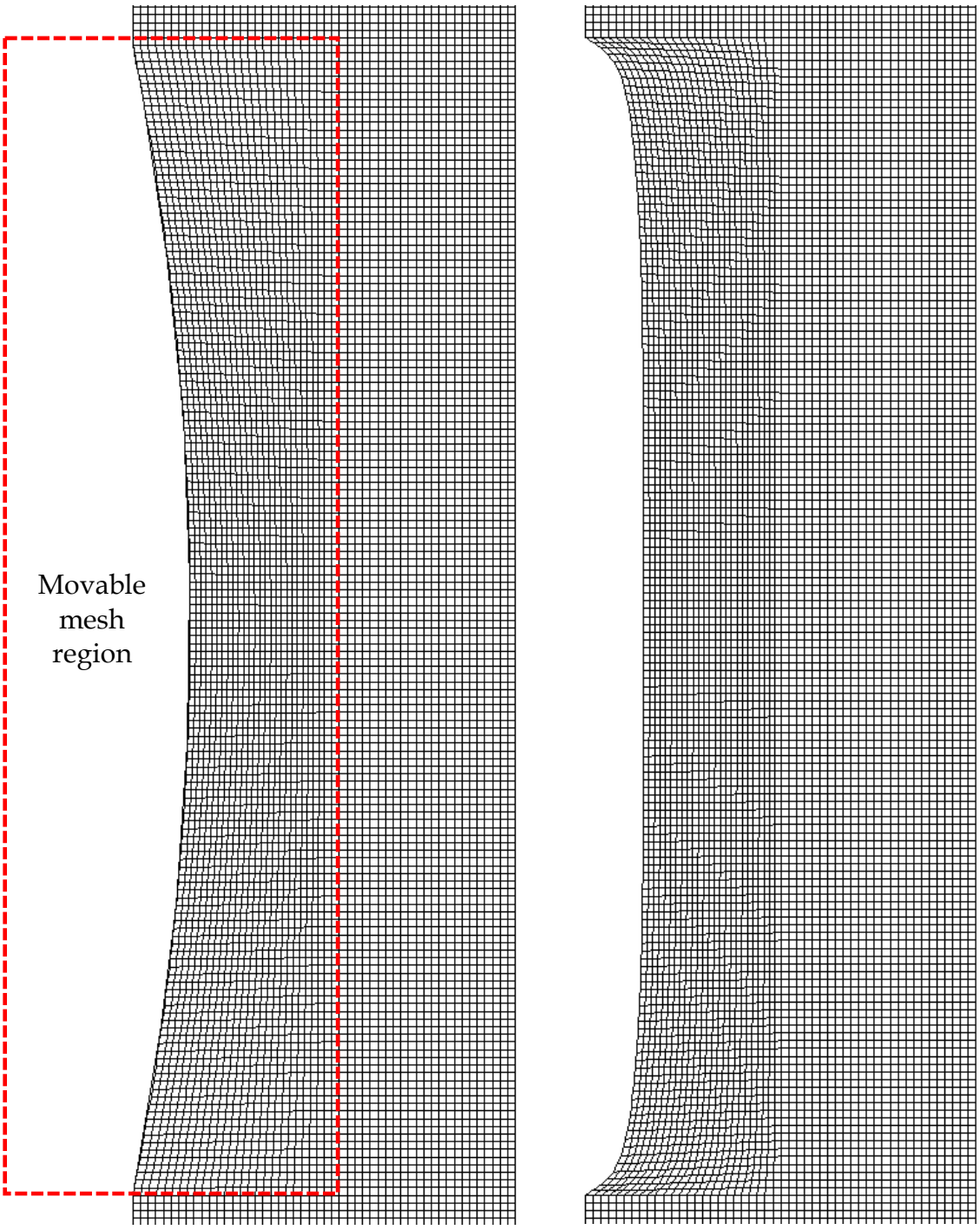


Figure 20: Initial mesh and shape (left) and final mesh and shape (right) for the open-boundary problem

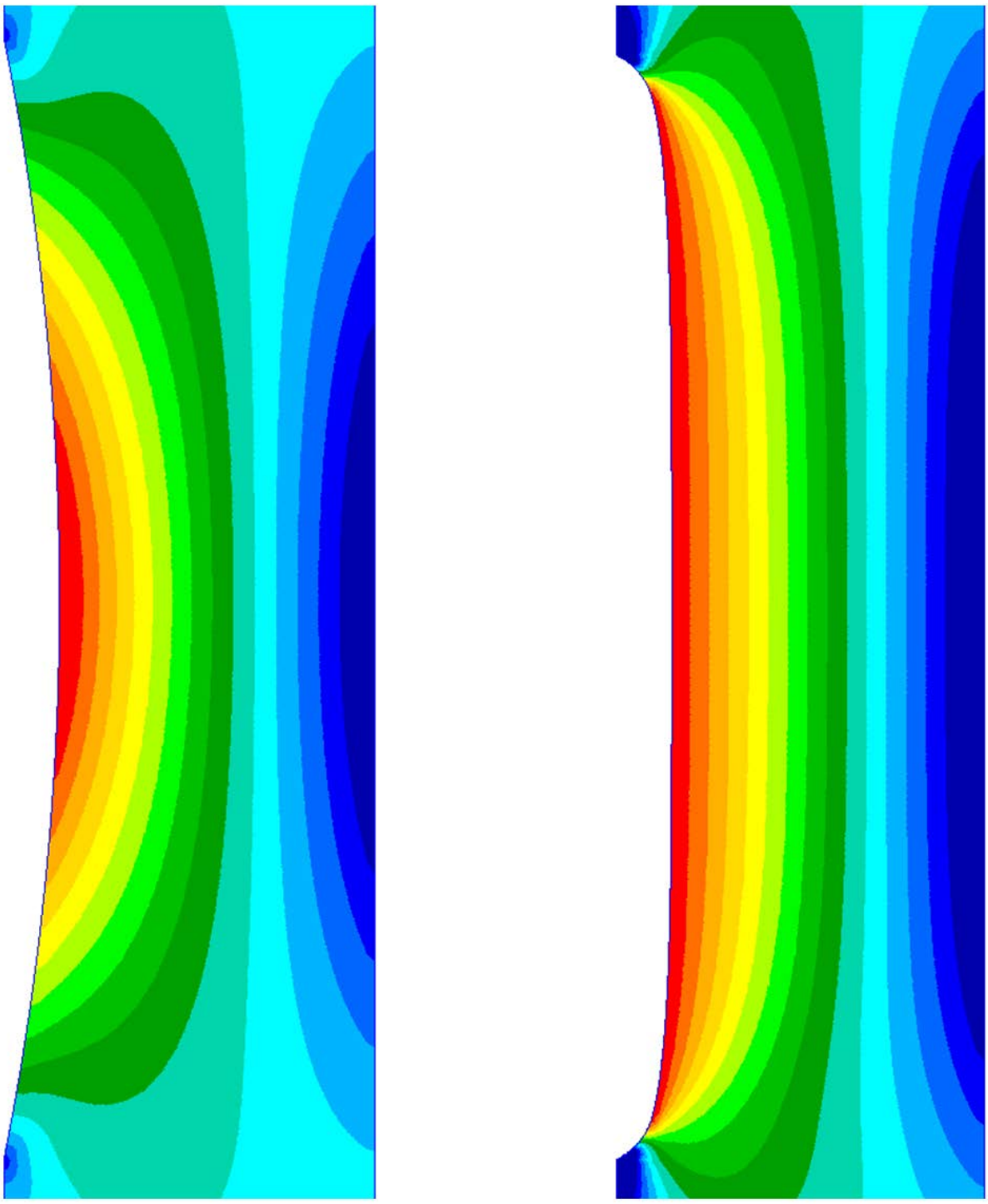


Figure 21: Stress contours of the major principal stress for the open-boundary problem obtained for the initial (left) and the final (right) fillet shapes

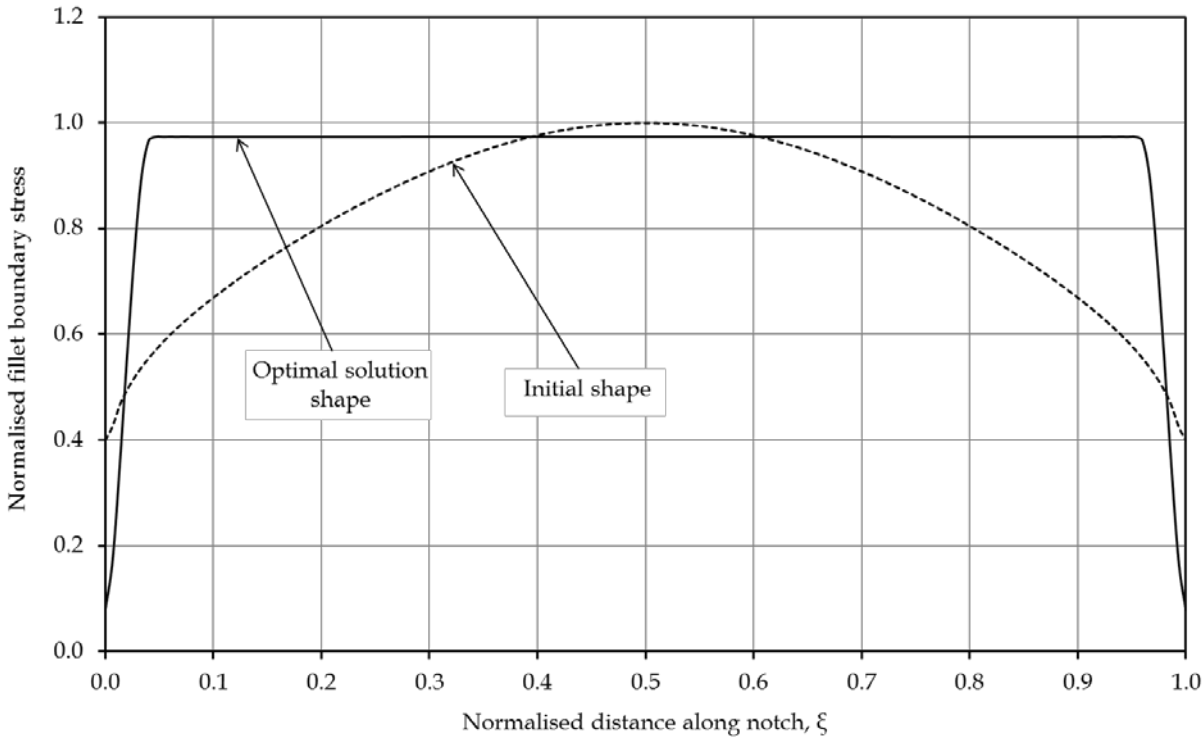


Figure 22: Comparison of the normalised boundary hoop stress for the initial notch shape and the final optimal notch shape

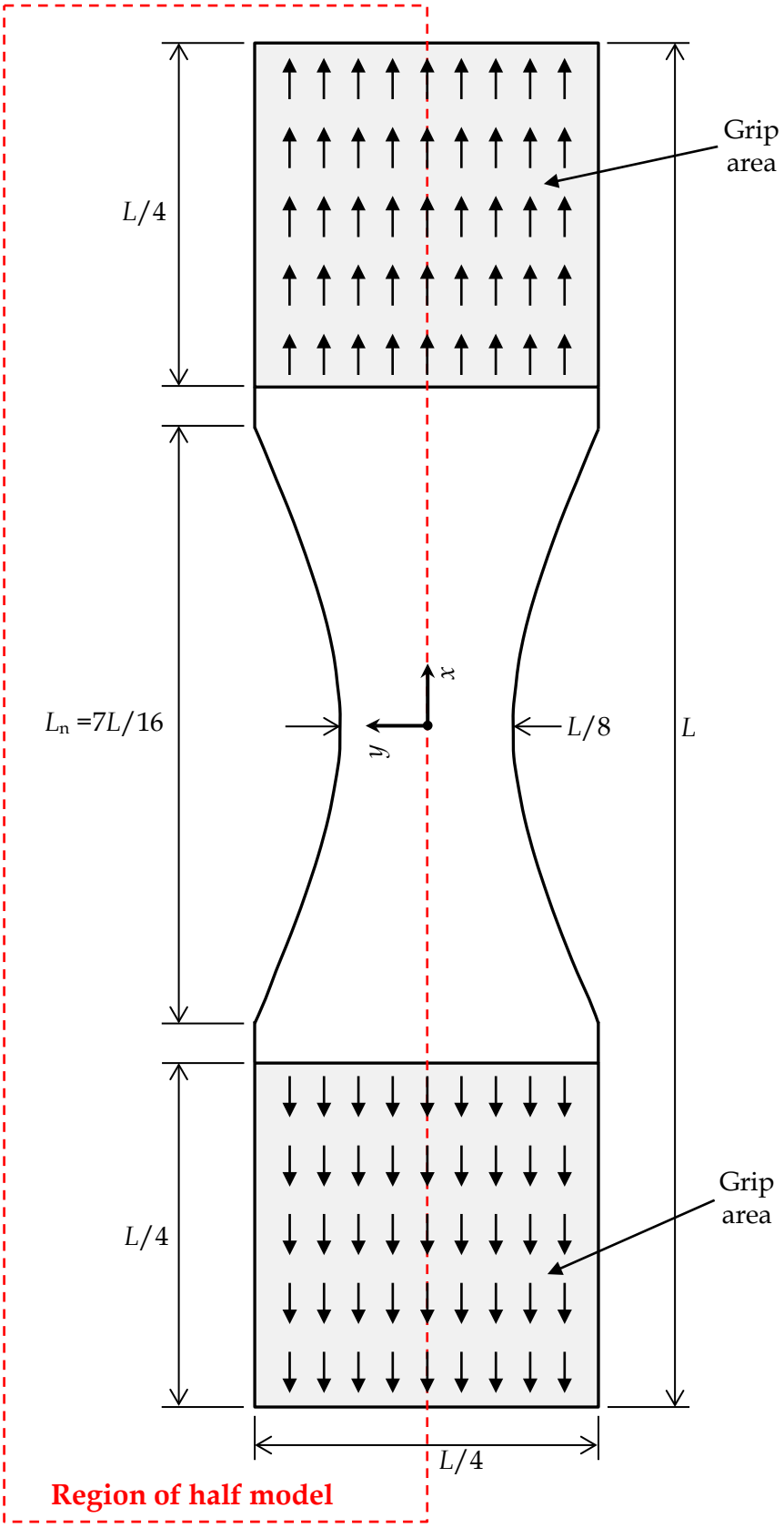


Figure 23: Geometry and loading arrangement for axial load test article modelled in 3D

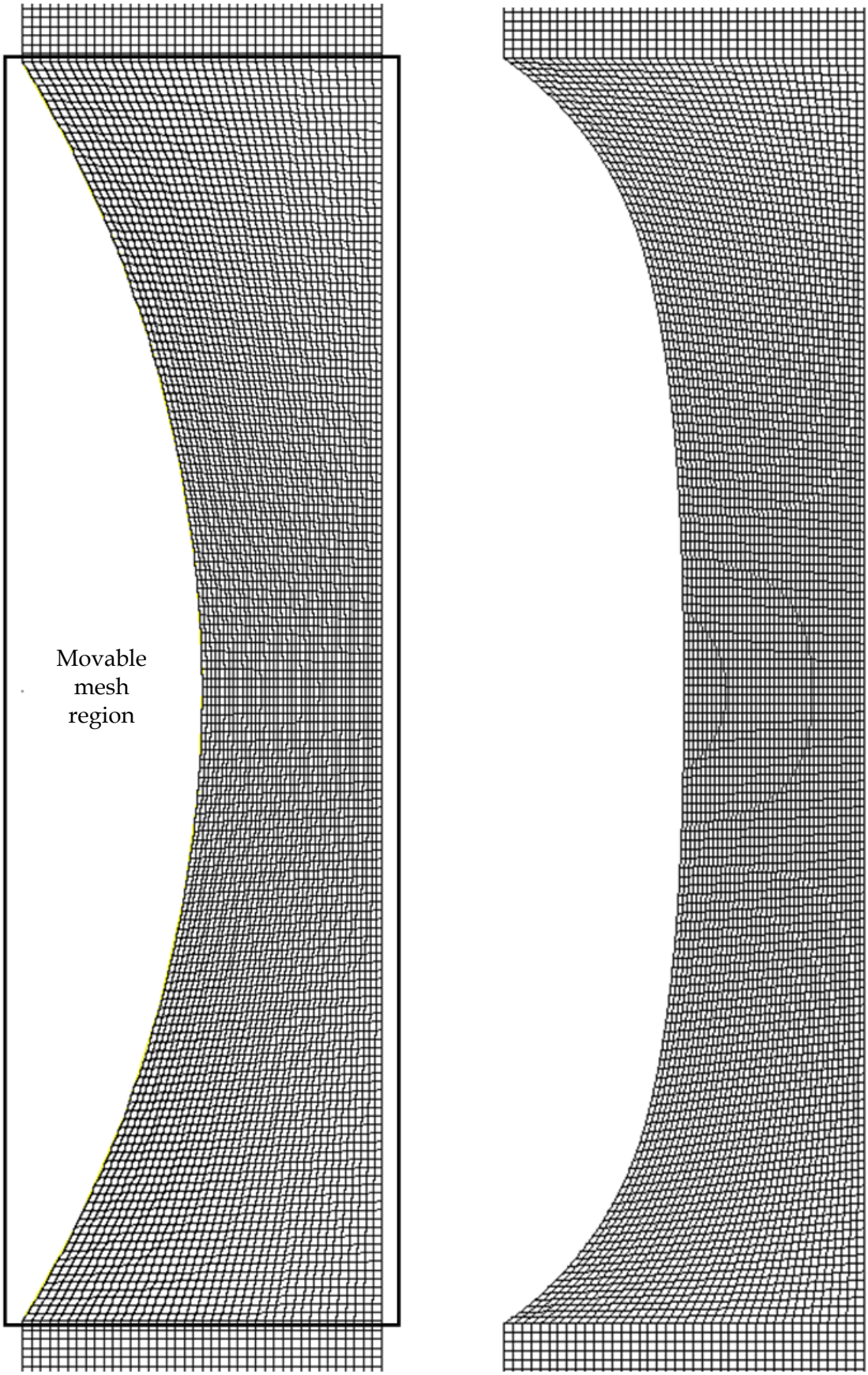


Figure 24: Plan view of initial (left) and final (right) finite element mesh patterns (semi-symmetric half model)

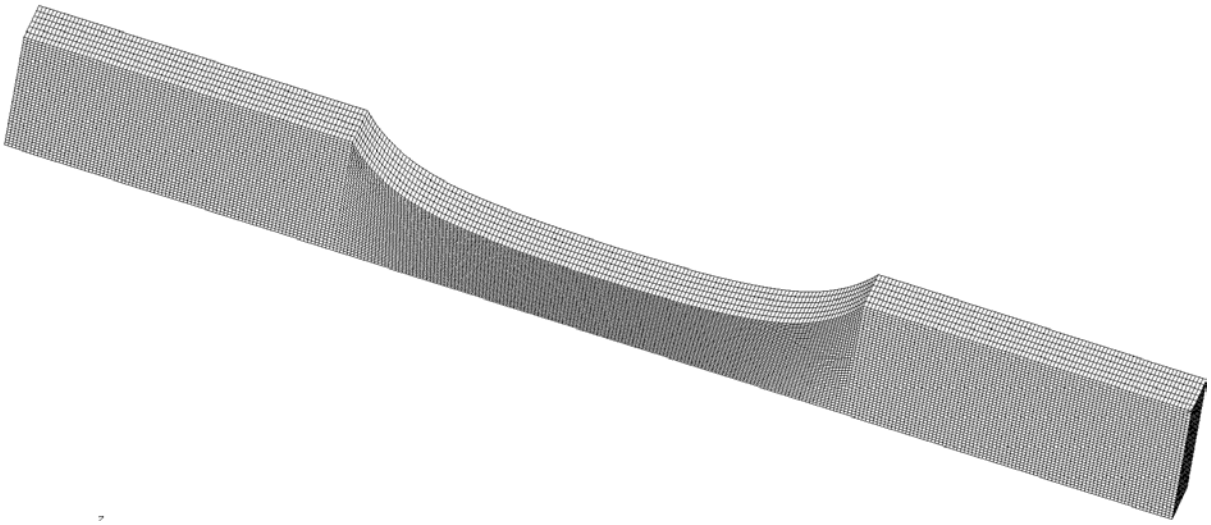


Figure 25: Isometric view of solution shape mesh (semi-symmetric half model)

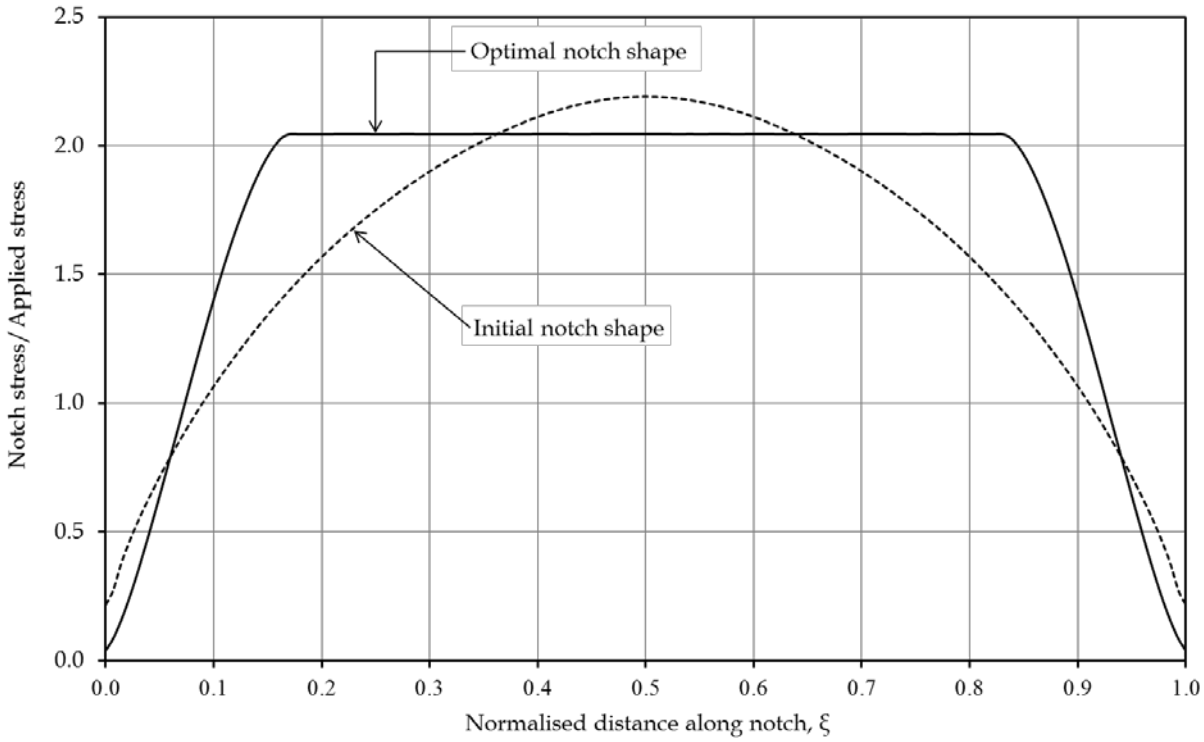


Figure 26: Normalised major principal stress along the surface of the notch (maximum value through the thickness)

Appendix A: Full listing of opscript.sh shell script

```
#!/bin/sh

echo ""
echo "================================================="
echo "  SHELL SCRIPT FOR RUNNING SHAPE OPTIMISATION JOBS"
echo "================================================="

time0=$(date +"%T")" "$(date +"%Y-%m-%d")
secs0=$(date +"%s")

echo ""
echo "Job start time: $time0"

# Run the installed Intel-supplied ifortvar.sh shell script
# to create the required environment variables for running the
# Intel FORTRAN compiler. The location of the shell script will
# depend on the specific version of the compiler.
#
# Note that these environment variables will remain local to
# the present shell as a result of using the "source" command.

echo ""
echo "Setting up Intel FORTRAN compiler environment to enable usage"
echo "of an Abaqus user subroutine with the shape optimisation job."

source /opt/intel/Compiler/11.1/069/bin/ifortvars.sh intel64

# Configure the files for performing the shape optimisation,
# as well as performing a cleanup.

cp nodes.ini nodes.dat
cp start.inp opjob.inp

rm -f convergence.dat
rm -f znodes???.dat
rm -f stress???.dat

# Perform the required number of iterations of the shape
# optimisation code.

maxitns=200

for (( i = 1;  i <= maxitns; i++ ))
do
  echo ""
  echo "====================================="
  echo "Starting iteration $i of $maxitns"
  echo "====================================="
  echo ""
  time1=$(date +"%T")" "$(date +"%Y-%m-%d")
  secs1=$(date +"%s")
  mv opjob.inp temp.inp
  rm -f opjob.*
```

```
  mv temp.inp opjob.inp
  abaqus job=opjob user=getsigq_ww cpus=1 interactive
  ./fsig    # Formats and collates stress???.dat files into stress.rpt
  ./rednf   # Reduces format of nodes.dat and puts in nodes.opt for optim4
  ./optim4  # Shape change C code. No mods have been done that affect function
  ./expnf   # Expands format of nodes.opt and puts into nodes.dat
  cp opjob.inp opjob.temp
  ./bdeckq  # Builds new Abaqus input deck using nodes.dat
  rm -f opjob.temp
  ./wrconv  # Writes peak hoop stress. Adds one line each iteration
  ./wrshape # Stores nodes.dat for each iteration in znodes???.dat
  time2=$(date +"%T")" "$(date +"%Y-%m-%d")
  secs2=$(date +"%s")
  etime=$(echo "scale=2;($secs2-$secs1)/60.0" | bc)
  echo ""
  echo "Iteration start time  : $time1"
  echo "Iteration finish time : $time2"
  echo "Iteration elapsed time: $etime minutes"
done

time3=$(date +"%T")" "$(date +"%Y-%m-%d")
secs3=$(date +"%s")
etime=$(echo "scale=3;($secs3-$secs0)/3600.0" | bc)

echo ""
echo "Job iterations  : $maxitns"
echo "Job start time  : $time0"
echo "Job finish time : $time3"
echo "Job elapsed time: $etime hours"
echo ""
echo "Shape optimisation job completed."
echo ""
```

Appendix B: Details of input data contained in file parameters.dat

Node File: must now be set to `nodes.ini`.

Surface File: Not used in this implementation.

Solid File: Not used in this implementation.

DB Name: Not used in this implementation.

Directory: path to working directory.

Length Mesh Seed: Not used in this implementation.

Through Mesh Seed: Not used in this implementation.

Equal Space File: Not used in this implementation.

Load Case File: Must be set to load.dat. Only the first line of the file is read.

Option 1: Multi-peak analysis.
Only works if set to true for both multi peak and single peak problems. `optim4` will fail if set to false.
Demonstrated by examples 1, 2 and 3.

Option 2: Read length mesh seed from file.
Not used in this implementation.

Option 3: Nodes defined in clockwise direction.
Only false option has been tested at this stage.
Demonstrated by examples 1, 2 and 3.

Option 4: Quasi-3D analysis.
Not used in this implementation.

Option 5: Apply equal spacing to boundary.
Must be set to true, demonstrated by examples 1, 2 and 3.

Option 6: Apply given through thickness 3D bias.
Not used in this implementation.

Option 7: Non-constant thickness 3D analysis.
Not used in this implementation.

Option 8: Apply multiple load cases – robust analysis.
Demonstrated by examples 1, 2 and 3.

Option 9: Spawn NASTRAN analysis independently – don't use analysis manager.
Not used in this implementation.

Option 10: Base input deck on entire model – otherwise only used current active group.
Not used in this implementation.

Option 11: Update LBCs – updates *z* constraints (TRUE) – only used for 3D analysis.
Not used in this implementation.

Option 12: Maintain integrity of initial shape – actively prevents boundary from crossing that of the initial profile.
Probably still works but not tested at this stage.

Option 13: Apply robust: perturbed analysis.
Demonstrated by example 2.

Option 14: Apply robust: independent analysis.
Demonstrated by example 2.

Option 15: Analysis is a restart – applies equal spacing on first iteration.
Not used in this implementation.

Option 16: Analysis has been started from command line – option to open database.
Not used in this implementation.

Option 17: Create mesh only – will only generate mesh, no analysis performed.
Not used in this implementation.

Option 18: Closed boundary problem.
Demonstrated by examples 1, 2 and 3.

Direction 1 min:
Direction 1 max:
Direction 2 min:
Direction 2 max:
These values probably still work as before. Not tested at this stage.

Property Set: Name of the property set to which the elements can be assigned.
Not used in this implementation.

Material: The material defined within PATRAN and associated with the property set defined above.
Not used in this implementation.

Thickness: Thickness of the model at the analysis point. This must be defined for both 2D and 3D analyses.
Not used in this implementation.

No Elem Through: Defined the number of elements through the thickness of the optimisation region.
Not used in this implementation.

Width Mesh Seed: The number of elements to be created across the defined surface. This currently set to 1 for most of the code.
Not used in this implementation.

Length Mesh Seed: This needs to be defined if the user does not supply a mesh seed in the 'mesh seed file' option. This defines the number of elements to be created along the length of the surface.
Not used in this implementation.

L2/L1: This defines the mesh seed bias through the thickness for a quasi-3D analysis.
Not used in this implementation.

Analysis Load Case: For models where there is more than 1 load case in the database, the load case to be used for the analysis is defined here.
Not used in this implementation.

1st Elem in Prop Region: As described above under property set, this is the first of the sequential elements in the property set to be updated.
Not used in this implementation.

Equiv Tolerance: Sets the equivalence tolerance for the mesh creation. Should typically be around 0.005–0.010.
Not used in this implementation.

Optim Threshold: This allows the user to define the method by which the stress threshold (σ_{th}) is determined.

The options are:

1.0 = peak stress in region
2.0 = average stress in region
3.0 = stress at the middle node in the region

These options probably still work as before, but remain untested at this stage.

Max Iter: Defines the maximum number of iterations to be used for the analysis.
This value is ignored. Number of iterations is determined by value in the Linux shell script.

Step Size: Defines the magnitude of the optimisation steps.
Demonstrated by examples 1, 2 and 3.

Min Rad: Defines the minimum radius of curvature for the analysis.
Demonstrated by examples 1, 2 and 3.

Analysis Plane: Defines the principal plane for the optimisation analysis. The options are "xy", "xz" and "yz" depending on the orientation of the model.
The movable mesh region of the model must be in the xy plane

Results Name: Defines the base name for the results file.
Not used in this implementation.

Delete Increment: Defines the frequency with which the results files will be retained.
Not used in this implementation.

Direction 1 centre: Centre of initial profile in direction 1 – used to check violation of initial shapes.
This value probably still works as before, but remains untested at this stage.

Direction 2 centre: Centre of initial profile in direction 2 – used to check violation of initial shapes.
This value probably still works as before, but remains untested at this stage.

Appendix C: Instructions for running 2D problems

This section constitutes interim documentation for the FORTRAN/Abaqus version of the shape optimisation code (2D).

This information applies as at 28 August 2014.

At this time, sample file sets, including all software, were located on the following Linux machine:

goated at `/home/StructDamMech/ShapeOpt/FtnAbaqus/...`

Files contained in the directories below the directory that is listed above (`.../fillet2d` and `.../hole2d`) are sample sets to use as a starting point to do an open-boundary problem (i.e. fillet) or a closed-boundary problem (i.e. hole). So the first step is to copy the appropriate entire directory to a working directory in your area. Set the permissions of all files to execute.

The term 3D is used throughout to indicate what has been called 2.5D or quasi-3D where the shape is 2D although the model is 3D (takes into account stress variation through-thickness).

Filenames are generally not variable. It may be good to leave it like this for clarity, i.e. for someone to take over a job from someone else.

The file sets include a sample problem relevant for that set. To create a problem different to the example problem the following steps are necessary:

1. Construct the model in a Patran file called `start.db`. The movable boundary must have a matching outer boundary with the same number of nodes. The mesh in between the inner and outer boundaries must be made up of regular 4-noded elements. The movable nodes include the intermediate nodes and must be on the x-y plane. Origin and other nodes can be anywhere.

2. In Patran create a 'node location report file' with the movable boundary nodes listed in path order (anticlockwise or from lower to upper). Remove text from top and put number of nodes on first line. Rename as `nodes.ini`.

3. Create a matching file of partner nodes which create a boundary for the movable mesh region, one partner node per movable node listed in the same path order. Call this file `pnodes.dat`.

4. Make sure all intermediate movable nodes (between movable boundary nodes and partner nodes lie on straight lines between their nearest boundary node and partner node.

5. Use Patran to write Abaqus input deck `patstart.inp`. Multiple linear static load cases can be written or just one as required. Output requests will later get over-written by `binid.f`.

6. Run `./addz`. This will put zeros in for the z nodal coordinates if no z values are present. Reads from `patstart.inp`. Writes to `allzstart.inp`.

7. Run `./binid`.

This routine will write from `allzstart.inp` to `start.inp` with a comment line after all nodes. This comment line has an integer value called nodetype, another integer value called nodepair and a third value (real) between 0 and 1 called posnode.

nodetype = 0 indicates the node lies outside the movable mesh region.

nodetype = 1 indicates the node on the line above is on the movable boundary.

nodetype = 2 indicates the node above is a partner node (i.e. in `pnodes.dat`).

nodetype = 3 indicates the node above lies between a boundary node and partner node.

Running the program `./counttype` will show how many nodes of each type are in `start.inp` for checking purposes.

nodepair indicates which pair of boundary node and partner node applies to the node above. This is the place in the list (not the node id). For example, nodepair=5 indicates that the 5th pair of boundary and partner nodes in `nodes.ini` and `pnodes.dat` are the relevant pair for the node on the line above.

posnode only applies to type 3 nodes and gives the position of the node along the line between its associated pair (as a fraction from the movable boundary outwards)

The program `binid` also deletes all output requests written by Patran and puts in a single request for principal stresses at nodes.

NB: Other more compact methods of handling posnode allowed for small errors to accumulate with each iteration, eventually creating mesh distortion. Keeping posnode fixed throughout the run was found to be helpful. Writing out the extra comment lines at each iteration did not significantly affect run time.

8. Edit the full path of the current directory (no spaces) in the file parameters.dat (line 5). The other data in this file is defined in the documentation for the PCL version created by Braemar [8]. This file can be mostly left unchanged. Some of this data is used by the program `optim4.c`. Some was used by the PCL code. The program `optim4.c` reads it all so it has been left unchanged at this stage.

9. Edit the path of the current directory (no spaces) to `getsig.f` line near the top that reads `path='...`

   This file is compiled when Abaqus runs so there is no need to compile separately.

10. Edit the number of boundary nodes and the number of load cases in the main routine of optim4.c Main routine is near the top of listing. Compile using `./compall.sh`. This shell script compiles and links all the FORTRAN routines and the single C routine.

11. Edit the number of load cases to the first line of `load.dat`.

12. Edit `opscript.sh` as required. That is, set the number of iterations in the `for` loop (it may be best to use 1 at first in order to see if the job runs as expected).

13. Run the job, i.e. `./opscript.sh`.

14. Open the file `convergence.dat` while the job is running to check on its progress. Use Ctrl-z to stop the job if necessary. You may have to wait for Abaqus to finish as it will continue to run a job that is running.

15. Create a new Patran database and read in `opjob.inp` to check the optimised shape.

Note:

All of the FORTRAN routines are the same for 2D fillet problems and 2D hole problems, except for editing the directory path contained in `getsig.f`. (3D versions of the code, where they differ from their 2D versions, have a `q` added at the end of the file name.)

Use `compall.sh` to compile and link all FORTRAN routines and the C routine. (This script can be used any time that the source code is changed.)

The program `optim4.c` is also the same for all 4 types of problems (2D and 3D) except for editing the number of movable boundary nodes and the number of load cases in its main routine at the near the top of the listing. It actually gets the number of movable nodes from elsewhere but still needs the number of load cases to be edited into the C source code at this stage.

Appendix D: Instructions for running 3D problems

This section constitutes interim documentation for the FORTRAN/Abaqus version of the shape optimisation code (3D).

This information applies as at 28 August 2014.

At this time sample file sets, including all software was located on the following Linux machine:

goated at `/home/StructDamMech/ShapeOpt/FtnAbaqus/...`

Files contained in the directories below the directory that is listed above (`.../fillet3d` and `.../hole3d`) are sample sets to use as a starting point to do an open-boundary problem (e.g. a fillet) or a closed-boundary problem (e.g. a hole). Copy the entire directory to a working directory on the Linux machine. Set permissions of all the executable files to execute. Maybe run a sample problem using `./opscriptq.sh`.

Filenames are generally not variable. It may be good to leave it like this for clarity, i.e. to enable someone to take over a job from someone else.

To create a problem that is different to the example problem, the following steps are necessary:

Note 1: The process is similar to the 2D cases. The FORTRAN routines with names ending with a `q` have been edited to cope with the additional nodes through the thickness. FORTRAN routines with names not ending in a q are the same as the 2D routines

Note 2: The master nodes are in the x-y plane at $z = 0$ (e.g. from 2D model used as a starting point). Slave nodes have the same x and y coordinates but vary in the z direction, i.e. several slave nodes for each master node depending on number of layers of nodes.

1. Construct the finite element model in the Patran database file `start.db`. The movable master nodes for the initial 2D model must be in the x-y plane at $z = 0$. Create the 3D model by extruding the 2D elements in the z direction to make 3D elements. Delete the 2D elements.

2. In Patran, create a 'node location report file' with the master movable boundary nodes listed in path order (anticlockwise or from lower to upper). Remove text from top and put number of master movable boundary nodes on the first line. Rename the file as `nodes.ini` (these are type 1 nodes).

3. Create a matching file of partner nodes which create a boundary for the movable mesh region, one partner node per master movable boundary node listed in the same path order. Call this file `pnodes.dat` (these are type 2 nodes).

4. Make sure all intermediate movable nodes (between movable boundary nodes and partner nodes lie on straight lines between their nearest boundary node and partner node.

5. Use Patran to write the Abaqus input deck `patstart.inp`. Multiple linear static load cases can be written or just one as required. Output requests will later get over-written by `binid1q.f`.

6. Run `./addz`. This will put zeros in for the z nodal coordinates if no z values are present. Reads from `patstart.inp`. Writes to `allzstart.inp`.

7. Run `./binid1q`.

   This routine will read from `allzstart.inp` and write to `type123.inp` inserting a comment line after all node lines. This comment line has an integer value called nodetype, another integer value called nodepair and a third value (real) between 0 and 1 called posnode.

   nodetype = 1 indicates that the node on the line above is on the master movable boundary.

   nodetype = 2 indicates that the node above is a master partner node (i.e. in `pnodes.dat`).

   nodetype = 3 indicates that the node above lies between a master boundary node and a master partner node

   nodetype = 0 indicates that the node is outside the movable mesh region (and could also be a slave node at this stage).

   nodepair indicates which pair of master boundary node and master partner node applies to the node above. This is the place in the list (not the node id). For example, nodepair=5 indicates that the 5th pair of boundary and partner nodes in `nodes.ini` and `pnodes.dat` are the relevant pair for the node on the line above.

   posnode only applies to type 3 master nodes and gives the position of the node along the line between its associated pair (as a fraction from the movable boundary outwards).

   The program `binid1q` also deletes all output requests written by Patran and puts in a single request for principal stresses to be computed at the nodes.

8. Run `./binid2q`. This routine will give an integer value node type to the slave nodes reading from `type123.inp` and writing to `start.inp`. It can take several minutes to run depending on model size. It will also give type 6 nodes the same posnode value as their matching type 3 node.

   nodetype = 4 indicates the node above is a slave to a type 1 node.

   nodetype = 5 indicates the node above is a slave to a type 2 node.

   nodetype = 6 indicates the node above is a slave to a type 3 node.

   The program `./counttype` can be run in order to count nodes by type for checking purposes.

9. Run `./binid3q`. It reads from `start.inp` and writes groups of node ids to `bndall.nid`. There is one group for each type1 node consisting of the type 1 node id followed by the ids of its slave nodes (type 4).

   The file `bndall.nid` is required by the `getsigq` routine that extracts the stresses from the Abaqus results.

10. Edit the full path of the current directory (no spaces) in the file `parameters.dat` (line 5). The other data in this file is defined in the documentation for the PCL version produced by Braemar [8]. This file should be left unchanged other than the path entry and the step size entry. Some of this data is used by `optim4.c`. Some was used by the PCL code. The program `optim4.c` reads it all so it has been left unchanged at this stage to avoid making modifications to `optim4.c`.

11. Edit the number of boundary nodes and the number of load cases in the main routine of `optim4.c`. The main routine is near the top of the listing. Compile the program by using the shell script `./compallq.sh`.

12. Write the number of load cases to the first line of `load.dat`.

13. Write the full path of the current directory (no spaces) to `getsigq.f`. This routine averages the nodal principal stresses and finds the maximum stress through the thickness.

    This file is compiled and run when Abaqus runs, so there is no need to compile it separately.

14. Edit `opscriptq.sh` as required. That is, set the number of iterations in the `for` loop, where it is best to use 1 at first to see if it runs. As a rough guide if step size is adjusted so that peak stress reduction at second iteration is about 0.1% problem usually converges in about 200 iterations. If additional iterations are required a restart can be executed by running just the loop and the statements within in `opscriptq.sh`.

15. Run the job by executing the shell script `./opscriptq.sh`. Watch the file `convergence.dat` for a while to see if the peak stress is coming down at a reasonable rate. It is possible to look at the mesh during run time by copying `opjob.inp` to a file with another name and reading it into a new Patran database.

16. Open the file `convergence.dat` while the job is running to check progress. Use Ctrl-z to stop job if necessary. You may have to wait for abaqus to finish running a job that was in progress.

17. Create a new Abaqus Patran data base and read in `opjob.inp` to check shape.

Notes:

All of the FORTRAN routines are the same for fillet problems and hole problems, except for editing the directory path contained in `getsigq.f`.

All the routines can be compiled using the shell script `./compallq.sh`, which needs to be done even if only a minor change is made to just one of the routines.

The program `optim4.c` is also the same for both types of problems, except for editing the number of movable boundary nodes and the number of load cases.

Appendix E: Source code listings of FORTRAN and C programs and shell scripts

Shell script compallq.sh

```
#!/bin/sh

echo ""
echo "================================================================="
echo "Compiling and linking programs needed for shape optimisation job."
echo "================================================================="
echo ""
echo "* indicates a program called from the shape optimisation script."
echo ""
echo "Setting up Intel Fortran compiler environment..."
source /opt/intel/Compiler/11.1/069/bin/ifortvars.sh intel64
echo ""

echo "Creating   addz..."
ifort addz.f -o addz
echo "Creating * bdeckq..."
ifort bdeckq.f -o bdeckq
echo "Creating   binid1q..."
ifort binid1q.f -o binid1q
echo "Creating   binid2q..."
ifort binid2q.f -o binid2q
echo "Creating   binid3q..."
ifort binid3q.f -o binid3q
echo "Creating   counttype..."
ifort counttype.f -o counttype
echo "Creating * expnf..."
ifort expnf.f -o expnf
echo "Creating * fsig..."
ifort fsig.f -o fsig
echo "Creating   gethoop..."
ifort gethoop.f -o gethoop
echo "Creating * rednf..."
ifort rednf.f -o rednf
echo "Creating * wrconv..."
ifort wrconv.f -o wrconv
echo "Creating * wrshape..."
ifort wrshape.f -o wrshape
echo "Creating * optim4..."
gcc -Wall optim4.c -lm -o optim4

echo ""
echo "Finished compiling and linking Fortran and C programs."
echo ""
```

Program addz.f

```
!=====================================================================

      program addz

! Puts zeros in for z-coordinate in Abaqus input deck if they are not
! already present in the data.
```

```
!
! Use double precision for (x,y,z) to ensure that we write out exactly
! what was read in (other than for z, which is set to zero).

      character aline*80,inpfile*80,outfile*80
      integer   nid
      real*8    x,y,z

      inpfile='patstart.inp'
      outfile='allzstart.inp'

      open(unit=20,file=inpfile)
      open(unit=30,file=outfile)

      write(*,*) 'Putting zeros in for z-coordinates...'
      write(*,*) 'Input file  = '//inpfile(1:len_trim(inpfile))
      write(*,*) 'Output file = '//outfile(1:len_trim(outfile))

      do
        read(20,100) aline
  100   format(a)
        write(30,100) aline
        if (aline(1:5).eq.'*NODE') go to 210
      end do

  210 continue

      do
        read(20,100)aline
        if (aline(1:2).eq.'**') then
          write(30,100) aline
          go to 220
        end if
        read(aline,*,iostat=ier) nid,x,y,z
        if (ier.ne.0) z=0.0d0
        write(30,500) nid,x,y,z
  500   format(i10,',',2x,g20.12,',',2x,g20.12,',',2x,g20.12)
      end do

  220 continue

      do
        read(20,100,iostat=ier)aline
        if (ier.ne.0) go to 200
        write(30,100)aline
      end do

  200 continue

      close(20)
      close(30)

      write(*,*) 'Finished addz.'

      stop
      end
```

Program bdeckq.f

```
!=====================================================================

      program bdeckq

! Builds a new Abaqus input deck with edited node locations based on
! nodes.dat.

      character aline*80,line1*80,line2*80
      dimension xnew(1000),ynew(1000)
      dimension px(1000),py(1000)

      open(unit=10,file='nodes.dat')
      open(unit=17,file='pnodes.dat')
      read(10,*) numnodes
      read(17,*)
      do i=1,numnodes
        read(10,*) nid,xnew(i),ynew(i)
        read(17,*) nid,px(i),py(i)
      end do
      close(10)
      close(17)

      open(unit=20,file='opjob.temp')
      open(unit=30,file='opjob.inp')

      do
        read(20,100) aline
        write(30,100) aline
  100   format(a)
        if (aline(1:5).eq.'*NODE') exit
      end do

      do
        read(20,100) line1
        read(20,100) line2
        if (line1(1:14).eq.'**end_of_nodes') then
          write(30,100) line1
          write(30,100) line2
          go to 210
        end if
        read(line1,500) n id,x,y,z
  500   format(i8,',',2x,g20.12,',',2x,g20.12,',',2x,g20.12)
        read(line2,501) nodetype,nodepair,posnode
  501   format('**mesh_data',2x,i5,2x,i5,g20.12)
        if (nodetype.eq.1 .or. nodetype.eq.4) then
          x=xnew(nodepair)
          y=ynew(nodepair)
          write(30,500) nid,x,y,z
          write(30,100) line2
        end if
        if (nodetype.eq.3 .or. nodetype.eq.6) then
          dpx=px(nodepair)-xnew(nodepair)
          dpy=py(nodepair)-ynew(nodepair)
          x=xnew(nodepair)+posnode*dpx
          y=ynew(nodepair)+posnode*dpy
          write(30,500) nid,x,y,z
          write(30,100) line2
```

```
        end if
        if (nodetype.eq.0 .or. nodetype.eq.2 .or. nodetype.eq.5) then
          write(30,500) nid,x,y,z
          write(30,100) line2
        end if
      end do

  210 continue

      do
        read(20,100,iostat=ier) aline
        if (ier.ne.0) exit
        write(30,100) aline
      end do

      close(20)
      close(30)

      write(*,*) 'Finished bdeck.'

      stop
      end
```

Program binid1q.f

```
!=====================================================================

      program binid1q

! Writes mesh data lines into the Abaqus input deck for type 1, 2 and
! 3 nodes. Also writes type 1, 2 and 3 nodes to file master.nodes.

      implicit none

      character aline*80
      real      nidbnd(1000),nidpar(1000)
      real      xbnd(1000),ybnd(1000),xpar(1000),ypar(1000)
      real      type3tol,posnode,x,y,z,dbnd,dpar,dtotal,derror
      integer   i,ier,icount,nid,nodetype,nodepair,numnodes

      ! This value can be adjusted so as to get a correct set of
      ! type 3 nodes.

      type3tol=0.0001

      write(*,*) 'Determining type 1, 2, and 3 nodes...'

      ! Read boundary nodes and partner nodes into arrays.

      write(*,*) 'Input file  = ','nodes.ini'
      write(*,*) 'Input file  = ','pnodes.dat'

      open(unit=10,file='nodes.ini')
      open(unit=15,file='pnodes.dat')

      read(10,*) numnodes
      read(15,*)
      do i=1,numnodes
        read(10,*) nidbnd(i),xbnd(i),ybnd(i)
```

```
        read(15,*) nidpar(i),xpar(i),ypar(i)
      end do

      close(10)
      close(15)

      ! Start reading through Abaqus input deck.

      write(*,*) 'Input file  = ','allzstart.inp'
      write(*,*) 'Output file = ','type123.inp'
      write(*,*) 'Output file = ','master.nodes'

      open(unit=20,file='allzstart.inp')
      open(unit=30,file='type123.inp')
      open(unit=40,file='master.nodes')

      ! Read through first part of file until start of nodes.

      do
        read(20,100) aline
  100   format(a)
        write(30,100) aline
        if (aline(1:5).eq.'*NODE') go to 200
      end do

  200 continue

      ! Start of reading through nodes.

      icount=0

      do
        read(20,*,iostat=ier) nid,x,y,z
        if (ier.ne.0) then
          write(30,110)
  110     format('**end_of_nodes')
          go to 210
        end if

        ! For each node, check against boundary nodes and partner
        ! nodes to determine the node type, to find pair association
        ! and to calculate posnode.

        nodetype=0
        nodepair=0
        posnode=0.0
        do i=1,numnodes
          if (nid.eq.nidbnd(i)) then
            nodetype=1
            nodepair=i
            goto 212
          end if
          if (nid.eq.nidpar(i)) then
            nodetype=2
            nodepair=i
            goto 212
          end if
          ! dbnd is distance between current node and boundary node.
          dbnd=sqrt((x-xbnd(i))**2+(y-ybnd(i))**2)
```

```
        ! dpar is distance between current node and partner node.
        dpar=sqrt((x-xpar(i))**2+(y-ypar(i))**2)
        ! dtotal is distance between boundary node and partner node.
        dtotal=sqrt((xbnd(i)-xpar(i))**2+(ybnd(i)-ypar(i))**2)
        ! derror will be close to zero if current node lies between
        ! boundary node and partner node.
        derror=abs((dbnd+dpar-dtotal)/dtotal)
        if (derror.le.type3tol .and. z.eq.0.0) then
          nodetype=3
          nodepair=i
          posnode=dbnd/dtotal
          icount=icount+1
        end if
      end do

212   continue

      ! Write nodes and mesh data to new Abaqus input deck.
      ! Count number of typed nodes (ie type 1 2 or 3).
      ! Write typed nodes to master.nodes file.

      if (nodetype.ne.0) then
        write(40,500) nid,x,y,z
500     format(i8,',',2x,g20.12,',',2x,g20.12,',',2x,g20.12)
        write(40,501) nodetype,nodepair,posnode
501     format('**mesh_data',2x,i5,2x,i5,g20.12)
      end if
      write(30,500) nid,x,y,z
      write(30,501) nodetype,nodepair,posnode

    end do
    ! End of reading and writing nodes.

210 continue

    ! Keep going through rest of input deck.

    do
      read(20,100,iostat=ier)aline
      if (ier.ne.0) go to 220
      ! Put in output request for principal stresses at nodes.
      if (aline(1:12).eq.'*TEMPERATURE') then
        write(30,100) aline
        write(30,100) '*EL FILE, DIR=YES, POS=NODES, FREQ=1'
        write(30,100) 'SP'
        do
          read(20,100)aline
          if (aline(1:9).eq.'*END STEP') go to 230
        end do
      end if
      ! End of output requests.
230   continue
      write(30,100) aline
    end do

220 continue

    close(20)
    close(30)
```

```
      close(40)

      ! Write number of typed nodes to screen so user can check.

      write(*,520) icount
  520 format(' Number of type 3 nodes found = ',i10)
      write(*,*)
     & 'If this number is incorrect, edit type3tol in binid1q.f.'
      write(*,*)
     & 'Typical effective range for type3tol is 0.1-0.00001.'
      write(*,*) 'Finished binid1q.'

      stop
      end
```

Program binid2q.f

```
!=====================================================================

      program binid2q

! Write mesh data lines in Abaqus input deck for type 4, 5 and 6 nodes.

      character aline*80
      integer   inode

      open(unit=10,file='type123.inp')
      open(unit=20,file='master.nodes')
      open(unit=30,file='start.inp')

      write(*,*) 'Writing mesh lines for type 4, 5 and 6 nodes...'
      write(*,*) 'This routine may take a while to run.'

      write(*,*) 'Input file  = ','type123.inp'
      write(*,*) 'Input file  = ','master.nodes'
      write(*,*) 'Output file = ','start.inp'

      do
        read(10,100) aline
        write(30,100) aline
  100   format(a)
        if (aline(1:5).eq.'*NODE') exit
      end do

      slavetol=0.1
      numslaves=0
      inode=0

      do
        read(10,*,iostat=ier) nid,x,y,z
        if (ier.ne.0) goto 210
        inode=inode+1
        read(10,501) nodetype,nodepair,posnode
  501   format('**mesh_data',2x,i5,2x,i5,g20.12)
        if (nodetype.eq.0) then
          do
            read(20,*,iostat=ier) nidm,xm,ym,zm
            if (ier.ne.0) goto 205
            read(20,501) mntype,mnpair,posmn
```

```
            xtol=slavetol
            ytol=slavetol
            if (x.gt.xm-xtol .and. x.lt.xm+xtol) then
              if (y.gt.ym-ytol .and. y.lt.ym+ytol) then
                nodetype=mntype+3
                nodepair=mnpair
                posnode=posmn
                numslaves=numslaves+1
                goto 205
              end if
            end if
          end do
        end if
  205   continue
        rewind(20)
        write(30,500) nid,x,y,z
  500   format(i8,',',2x,g20.12,',',2x,g20.12,',',2x,g20.12)
        write(30,501) nodetype,nodepair,posnode
        if (mod(inode,5000).eq.0 .and. inode.ge.5000) then
          write(*,*) 'Done nodes  = ',inode
        end if
      end do

  210 continue
      write(*,*) 'Done nodes  = ',inode
      write(30,520)
  520 format('**end_of_nodes')

      do
        read(10,100,iostat=ier) aline
        if (ier.ne.0) exit
        write(30,100) aline
      end do

      close(10)
      close(20)
      close(30)

      write(*,*) 'Number of slave nodes found = ',numslaves
      write(*,*) 'If this number is incorrect, '//
     &           'adjust slavetol in binid2q.f.'
      write(*,*) 'Finished binid2q.'

      stop
      end
```

Program binid3q.f

```
!======================================================================

      program binid3q

! Write groups of node ids to bndall.nid, one group per type 1 node,
! all nodes through thickness (with same x and y coordinates).

      character aline*80
      dimension nidbnd(400,20)

      write(*,*) 'Writing groups of node ids...'
```

```
      write(*,*) 'Input file  = ','nodes.ini'
      write(*,*) 'Input file  = ','start.inp'
      write(*,*) 'Output file = ','bndall.nid'

      open(unit=10,file='nodes.ini')
      open(unit=20,file='start.inp')
      open(unit=30,file='bndall.nid')

      do i=1,400
        do j=1,20
          nidbnd(i,j)=0
        end do
      end do

      read(10,*) numnodes
      do i=1,numnodes
        read(10,*) nidbnd(i,1)
      end do
      close(10)

      do
        read(20,100) aline
  100   format(a)
        if (aline(1:5).eq.'*NODE') go to 200
      end do

  200 continue

      do
        read(20,*,iostat=ier) nid,x,y,z
        if (ier.ne.0) goto 210
        read(20,501) nodetype,nodepair,posnode
  501   format('**mesh_data',2x,i5,2x,i5,g20.12)
        if (nodetype.eq.4) then
          do j=2,20
            if (nidbnd(nodepair,j).eq.0) then
              nidbnd(nodepair,j)=nid
              goto 205
            end if
          end do
        end if
  205   continue
      end do

  210 continue

      do i=1,numnodes
        do j=1,20
          if (nidbnd(i,j).ne.0) write(30,*) nidbnd(i,j)
        end do
        write(30,510)
  510   format('end_of_group')
      end do

      close(20)
      close(30)

      write(*,*) 'Finished binid3q.'
```

```
      stop
      end
```

Program countsn.f

```
!=====================================================================

      program countsn

! Counts nodes by type for checking purposes.

      character aline*80

      open(unit=10,file='start.inp')

      num0=0
      num1=0
      num2=0
      num3=0
      num4=0
      num5=0
      num6=0

      do
        read(10,100,iostat=ier) aline
  100   format(a)
        if (ier.ne.0) goto 200
        if (aline(1:11).eq.'**mesh_data') then
          read(aline,500) nodetype
  500     format(11x,i7)
          if (nodetype.eq.0) num0=num0+1
          if (nodetype.eq.1) num1=num1+1
          if (nodetype.eq.2) num2=num2+1
          if (nodetype.eq.3) num3=num3+1
          if (nodetype.eq.4) num4=num4+1
          if (nodetype.eq.5) num5=num5+1
          if (nodetype.eq.6) num6=num6+1
        end if
      end do

  200 continue

      write(6,505) num0
  505 format('Number of type 0 nodes found = ',i10)
      write(6,510) num1
  510 format('Number of type 1 nodes found = ',i10)
      write(6,520) num2
  520 format('Number of type 2 nodes found = ',i10)
      write(6,530) num3
  530 format('Number of type 3 nodes found = ',i10)
      write(6,540) num4
  540 format('Number of type 4 nodes found = ',i10)
      write(6,550) num5
  550 format('Number of type 5 nodes found = ',i10)
      write(6,560) num6
  560 format('Number of type 6 nodes found = ',i10)

      close(10)
```

```
      stop
      end
```

Program cpxy.f

```
!=====================================================================

      program cpxy

C Puts first line and first column back in to nodes.dat

      open(unit=10,file='znodes400.dat')
      open(unit=20,file='deepercutv2.xyz')

      read(10,*) numnodes
      do i=1,numnodes
        read(10,*) nid,x,y,z
        xnew=y-80
        ynew=20-x
        write(20,500) xnew,ynew
  500   format(f20.7,f20.7)
      end do

      stop
      end
```

Program expnf.f

```
!=====================================================================

      program expnf

! Puts first line and first column back into nodes.dat.

      integer nid(10000)

      ! Read in all of node numbers in preparation for writing them out.

      open(unit=10,file='nodes.dat')
      read(10,*) numnodes
      do i=1,numnodes
        read(10,*) nid(i)
      end do
      close(10)

      ! Read in the shape coordinates obtained from the current iteration,
      ! and write them out together with the node numbers.

      open(unit=10,file='nodes.dat')
      open(unit=20,file='nodes.opt')

      read(20,*) numnodes
      write(10,'(i8)') numnodes
      do i=1,numnodes
        read(20,*) j,x,y,z
        write(10,'(i8,4f12.6)') nid(i),x,y,z
      end do

      close(10)
```

```
      close(20)

      write(6,*) 'Finished expnf.'

      stop
      end
```

Program fsig.f

```
!=====================================================================

      program fsig

! Collates stress files from getsig (1 per load case) into one file
! called stress.rpt. Formats data for use by the optim4 program.

      character fname*12, loadstr*2

      open(unit=10,file='load.dat')
      read(10,*)numloads
      close(10)

      do i=1,numloads
        write(loadstr,500) i
  500   format(i2)
        if (loadstr(1:1).eq.' ') loadstr(1:1)='0'
        fname(1:6)='stress'
        fname(7:8)=loadstr
        fname(9:12)='.txt'
        open(unit=i+10,file=fname,status='OLD')
      end do

      open(unit=9,file='stress.rpt')

      do
        do i=1,numloads
          read(i+10,*,iostat=ier) nodeid,s11,s22
          if (ier.ne.0) go to 600
          write(9,510) i,nodeid,s11,s22
  510     format(i2,2x,i8,2x,g20.12,2x,g20.12)
        end do
      end do

  600 continue

      do i=1,numloads
        close(i+10)
      end do

      close(9)

      write(6,*) 'Finished fsig.'

      stop
      end
```

Program gethoop.f

```
!======================================================================

      program gethoop

! Gets hoop stress from stress??.txt file and puts in single column for
! plotting.

      open(unit=10,file='stress03.txt')
      open(unit=20,file='zero.hoop')

      i=0
      do
        i=i+1
        read(10,*,iostat=ier) nid,s11,s22
        if (ier.ne.0) go to 200
        hoop=s11
        if (abs(s22).ge.s11) hoop=s22
        write(20,*) i,hoop
      end do

  200 continue

      close(10)
      close(20)

      stop
      end
```

Subroutine urdfil.f

```
!======================================================================

      subroutine urdfil(lstop,lovrwrt,kstep,kinc,dtime,time)

! Abaqus user-defined subroutine to get principal stresses at nodes
! from the .fil results file. Averages all stress values given for
! each boundary node and finds the maximum value through the
! thickness (z direction).
!
! This file is compiled and run by Abaqus.

      include 'ABA_PARAM.INC'

      dimension array(513),jrray(nprecd,513),time(2)
      dimension nidbnd(300,10)
      dimension s11(300,10,10),s22(300,10,10),numsigs(300,10)
      dimension totals11(300,10),totals22(300,100)
      dimension avs11(300,10),avs22(300,10)
      dimension avmaxs11(300),avmaxs22(300)
      character stepstr*2,fname*12,flocn*80,path*68,aline*80

      equivalence (array(1),jrray(1,1))

      call posfil(kstep,kinc,array,jrcd)

!     open(unit=2500,file='/home/waldmanw/abaqus/abopt/fillet3dmod/debug.txt')
```

```
      path='/home/waldmanw/abaqus/abopt/fillet3dmod/'
      lenpath=len_trim(path)

      ! i is number of boundary node.
      ! j is layer number through thickness, and j=1 indicates
      !   boundary master nodes.
      ! k is number of s11 values found in results file for node
      !   nidbnd(i,j).

      ! Count number of layers of nodes.

      open(unit=2120,file=path(1:lenpath) // 'bndall.nid')
      n=0
      do
        n=n+1
        read(2120,*,iostat=ier) nidtemp
        if (ier.ne.0) then
          nlayers=n-1
          go to 118
        end if
      end do
  118 continue
      rewind(2120)

      ! Read boundary node nids into array nidbnd, columns 1 - nlayers.

      i=0
      do
        i=i+1
        do j=1,nlayers
          read(2120,*,iostat=ier) nidbnd(i,j)
          if (ier.ne.0) goto 135
        end do
        read(2120,*)
      end do
  135 continue
      numnodes=i-1
      close(2120)

      ! Open file of form stress??.txt, where ?? is the loadcase (step)
      ! number in the range 01-99.

      write(stepstr,500) kstep
  500 format(i2)
      if (stepstr(1:1).eq.' ') stepstr(1:1)='0'
      fname(1:6)='stress'
      fname(7:8)=stepstr
      fname(9:12)='.txt'
      flocn=path(1:lenpath)//fname

      open(unit=2130,file=flocn)

      ! Initialise numsigs with zeros. Numsigs is an array of counters
      ! that count s11 values.

      do i=1,numnodes
        do j=1,nlayers
          numsigs(i,j)=0
        end do
```

```
      end do

      ! Read through all records in the Abaqus results file.

      do
        call dbfile(0,array,jrcd)
        if (jrcd.ne.0) go to 110
        key=jrray(1,2)
        if (key.eq.1) then
          lemid=jrray(5,1)
          nodeid=jrray(7,1)
        end if
        if (key.eq.401) then
          sig11=array(5)
          sig22=array(3)
        end if
        if (key.eq.401) then
          do i=1,numnodes
            do j=1,nlayers
              if (nodeid.eq.nidbnd(i,j)) then
                numsigs(i,j)=numsigs(i,j)+1
                s11(i,j,numsigs(i,j))=sig11
                s22(i,j,numsigs(i,j))=sig22
              end if
            end do
          end do
        end if
      end do

  110 continue

!     do i=1,numnodes
!       do j=1,nlayers
!         do k=1,numsigs(i,j)
!           write(2500,*) nidbnd(i,j),i,j,k
!           write(2500,*) s11(i,j,k),s22(i,j,k)
!           write(2500,*)
!         end do
!       end do
!     end do

      do i=1,numnodes
        do j=1,nlayers
          totals11(i,j)=0.0
          totals22(i,j)=0.0
          do k=1,numsigs(i,j)
            totals11(i,j)=totals11(i,j)+s11(i,j,k)
            totals22(i,j)=totals22(i,j)+s22(i,j,k)
          end do
          avs11(i,j)=totals11(i,j)/numsigs(i,j)
          avs22(i,j)=totals22(i,j)/numsigs(i,j)
        end do
      end do

      do i=1,numnodes
        avmaxs11(i)=0.0
        avmaxs22(i)=0.0
        do j=1,nlayers
          if (abs(avs11(i,j)).gt.abs(avmaxs11(i))) then
```

```
            avmaxs11(i)=avs11(i,j)
          end if
          if (abs(avs22(i,j)).gt.abs(avmaxs22(i))) then
            avmaxs22(i)=avs22(i,j)
          end if
        end do
        write(2130,510) nidbnd(i,1),avmaxs11(i),avmaxs22(i)
  510   format(i6,2x,g20.12,2x,g20.12)
      end do

  125 continue

      close(2130)

!     close(2500)

      return
      end
```

Program rednf.f

```
!=======================================================================

      program rednf

! Removes first line and first column from nodes.dat. Puts data in
! nodes.opt for optim4 program.

      open(unit=10,file='nodes.dat')
      open(unit=20,file='nodes.opt')

      read(10,*) numnodes
      do i=1,numnodes
        read(10,*) nid,x,y,z
        c=0.0
        write(20,'(4f12.6)') x,y,z,c
      end do

      close(10)
      close(20)

      write(*,*) 'Finished rednf.'

      stop
      end
```

Program wrconv.f

```
!=======================================================================

      program wrconv

! Write peak boundary hoop stress in convergence.dat, which can be
! viewed during run time to monitor job. After job is finished this
! file can used to create convergence plot.

      open(unit=10,file='stress.rpt')
      s11max=0.0
      do
```

```
        read(10,*,iostat=ier) lc,nid,s11,s22
        if (ier.ne.0) exit
        if (s11.gt.s11max) s11max=s11
      end do
      close(10)

      open(unit=20,file='convergence.dat',access='append')
      write(20,110) s11max
  110 format('Peak tensile hoop stress = ',g20.12)
      close(20)

      write(*,*) 'Finished wrconv.'

      stop
      end
```

Program wrshape.f

```
!=====================================================================

      program wrshape

! Stores nodes.dat at each iteration in znodes???.dat so that the
! shape at any iteration can be retrieved.

      character aline*80, stri*3, fname*13

      open(unit=10,file='convergence.dat')

      i=0
      do
        i=i+1
        read(10,100,iostat=ier) aline
  100   format(a)
        if (ier.ne.0) then
          itnum=i-1
          goto 200
        end if
      end do

  200 continue
      close(10)

      open(unit=11,file='nodes.dat')
      open(unit=12,file='stress01.txt')

      write(stri,110) itnum
  110 format(i3)
      if (stri(1:1).eq.' ') stri(1:1)='0'
      if (stri(2:2).eq.' ') stri(2:2)='0'

      fname(1:6)='znodes'
      fname(7:9)=stri
      fname(10:13)='.dat'
      open(unit=20,file=fname)

      fname(1:6)='stress'
      fname(7:9)=stri
      fname(10:13)='.dat'
```

```
      open(unit=21,file=fname)

      read(11,*) numnodes
      write(20,*) numnodes
      do i=1,numnodes
        read(11,100) aline
        write(20,100) aline
      end do

      do
        read(12,100,iostat=ier) aline
        if (ier.ne.0) goto 210
        write(21,100) aline
      end do

  210 continue

      close(11)
      close(12)
      close(20)
      close(21)

      write(6,*) 'Finished wrshape.'

      stop
      end
```

Program optim4.c

```
/*

Shape Optimisation Functions
============================

Written by R Braemar 06/12/2005.

Performs the optimisation portion of the DSTO/MSC Shape Optimisation
functions.

Reads stress and node data and calculates and applies node movements
accordingly.

Modified by R Kaye in 2013 to run on Linux machine.

Modifications do not affect functioning of the program.

Modified by W Waldman in 2014.

*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define PI 3.14159265358979323846
#define angle_mod 57.2957795130823208768

FILE *file_open(char *);

struct node_data
```

```
{
  double x;       /* x coordinate */
  double y;       /* y coordinate */
  double z;       /* z coordinate */
  double n_x;     /* previous x coordinate */
  double n_y;     /* previous y coordinate */
  double n_z;     /* previous z coordinate */
  double x_0;     /* original x */
  double y_0;     /* original y */
  double z_0;     /* original z */
  double con;     /* constraint */
  double normal;  /* angle of normal */
  double det;     /* determinant */
  double rad;     /* radius */
  double space_r; /* node_spacing ratio */
  double mp1;     /* modification parameter for given node */
  double thresh;  /* thresholh stress for node */
  double d;       /* optimisation factor d */
  int x_over;     /* 1 indicates a multipeak cross over point */
};

struct ana_options
{
  int num_node;          /* number of nodes on optimisation boundary */
  double min_rad;        /* minimum radius of curvature constraint */
  int x_y_z;             /* analysis plane */
  double step_size;      /* step size parameter s */
  int options[19];       /* flags options for analysis */
  double centroid[2];    /* centre point of initial shape, used for boundary calcs
*/
  double constraints[4]; /* the geometry constraints in the 2D plane, ie min x,
max x, min y, max y */
  double mesh_param[5];  /* thickness, no. elem through thickness, width mesh
seed, length mesh seed, opt threshold */
  int num_loadcase;
  char initial_filename[128];
};

struct stress
{
  double s1;
  double s2;
};

int main()
{
  FILE *node_data_file;
  FILE *stress_data_file;
  FILE *parameter_data_file;
  FILE *initial_data_file;
  FILE *node_ini_file;
  struct node_data *data;
  struct ana_options *parameter;
  int iteration;
  char fname_ini[]       ="nodes.ini";
  char fname_nodes[]     ="nodes.opt";
  char fname_stress[]    ="stress.rpt";
  char fname_parameters[]="parameters.dat";
```

```
  int read_data(FILE *, FILE *, FILE *, struct node_data *, struct ana_options *);
  void read_parameters(FILE *, struct ana_options *);
  void do_optimise_move(struct node_data *, struct ana_options *);
  void do_bound_mods(struct node_data *, struct ana_options *);
  void output_results(struct node_data *, struct ana_options *);
  void modify_z(struct node_data *data, struct ana_options *parameter);

  iteration = 1;

  /* read in geometry and stress data for each node */

  printf("Starting analysis.\n");

  /* open data files */

  node_data_file = fopen(fname_nodes,"r");
  stress_data_file = fopen(fname_stress,"r");
  parameter_data_file = fopen(fname_parameters,"r");

  if (parameter_data_file == NULL)
    {
      printf("Can't open parameter file: %s\n", fname_parameters);
      exit(1);
    }
  else if (node_data_file == NULL)
    {
      printf("Can't open node data file: %s\n", fname_nodes);
      exit(1);
    }
  else if (stress_data_file == NULL)
    {
      printf("Can't open stress data file: %s\n", fname_stress);
      exit(1);
    }

  printf("Reading optimisation analysis parameters from file.\n");
  printf("Parameters data file is: %s\n", fname_parameters);

  parameter = (struct ana_options *) calloc(1, sizeof(struct ana_options));

  /* read analysis parameters. 31 parameters to be read, contained in parameter
data file*/

  read_parameters(parameter_data_file, parameter);

  // Switch off smoothing of optimisation displacements at cross over region.

  parameter[0].options[18] = 1;

  initial_data_file = fopen(parameter[0].initial_filename,"r");
  if (initial_data_file == NULL)
    {
      printf("Can't open initial data file: %s\n", parameter[0].initial_filename);
      exit(1);
    }

  parameter[0].num_loadcase = 1;
  parameter[0].num_node = 81;
```

```
  node_ini_file = fopen(fname_ini,"r");
  if (node_ini_file == NULL)
    {
      printf("Can't open initial node file: %s\n", fname_ini);
      exit(1);
    }
  else
    {
      printf("Reading number of nodes from initial node file: %s\n",fname_ini);
      fscanf(node_ini_file, "%i\n", &parameter[0].num_node);
      fclose(node_ini_file);
      printf("Number of nodes to be processed = %d\n", parameter[0].num_node);
    }

  /* allocate struct size for data storage */

  data = (struct node_data *) calloc(parameter[0].num_node + 1, sizeof(struct
node_data));

  /* read remaining data */

  printf("Reading data and determining optimisation factors.\n");

  read_data(node_data_file, stress_data_file, initial_data_file, data, parameter);

  printf("All data read and optimisation factors determined.\n");

  /* close input files */

  fclose(parameter_data_file);
  fclose(node_data_file);
  fclose(stress_data_file);
  fclose(initial_data_file);

  /* do optimisation movements */

  do_optimise_move(data, parameter);

  /* perform radius calculations */

  do_bound_mods(data, parameter);

  printf("Finished boundary constraint modifications.\n");

  /* perform z modifications if required */

  if (parameter[0].options[6] == 1)
    modify_z(data, parameter);

  /* write output file */

  printf("Starting to write data.\n");

  output_results(data, parameter);

  /* clear memory */

  free(data);
  free(parameter);
```

```
  /*  printf("Optimisation analysis complete for iteration %d.\n", iteration); */

  return 0;
}

/*
*****************************************************
*/

void read_parameters(FILE *fp, struct ana_options *parameter)
{
  /* read in analysis parameters from file
     Parameters are stored in the specified struct.
  */

  int i, j;
  char *fgets(), temp_val_1[256], temp_val_2[256], temp_val_3[256];
  char logical[] = "TRUE", xy[] = "xy", xz[] = "xz", yz[] = "yz";

  for (j = 0; j < 4; j++)
    {
      if (j == 0)
        {
          for (i = 0; i < 9; i++)
            {
              if (i == 0)
                fscanf(fp, "%s %s %s\n", temp_val_1, temp_val_2,
parameter[0].initial_filename);
              else
                fgets(temp_val_1, 256, fp);
            }
        }
      else if (j == 1)
        {
          for (i = 0; i < 18; i++)
            {
              fscanf(fp, "%s %s %s\n", temp_val_1, temp_val_2, temp_val_3);
              if (*temp_val_3 == *logical)
                parameter[0].options[i] = 1;
              else
                parameter[0].options[i] = 0;
              printf("Option %02d: %d\n", i+1,parameter[0].options[i]);
            }
        }
      else if (j == 2)
        {
          for (i = 0; i < 4; i++)
            {
              fscanf(fp, "%s %s %s %lf\n", temp_val_1, temp_val_2, temp_val_3,
&parameter[0].constraints[i]);
            }
        }
      else if (j == 3)
        {
          fgets(temp_val_1,256,fp);
          fgets(temp_val_1,256,fp);

          fscanf(fp, "%s %lf\n", temp_val_1, &parameter[0].mesh_param[0]);
```

```
          fscanf(fp, "%s %s %s %lf\n", temp_val_1, temp_val_2, temp_val_3,
&parameter[0].mesh_param[1]);
          fscanf(fp, "%s %s %s %lf\n", temp_val_1, temp_val_2, temp_val_3,
&parameter[0].mesh_param[2]);
          fscanf(fp, "%s %s %s %lf\n", temp_val_1, temp_val_2, temp_val_3,
&parameter[0].mesh_param[3]);

          fgets(temp_val_1,256,fp);
          fgets(temp_val_1,256,fp);
          fgets(temp_val_1,256,fp);
          fgets(temp_val_1,256,fp);

          fscanf(fp, "%s %s %lf\n", temp_val_1, temp_val_2,
&parameter[0].mesh_param[4]);

          for (i = 0; i < 5; i++)
              printf("Mesh Parameter %d: %lf \n", i+1,parameter[0].mesh_param[i]);

          fgets(temp_val_1,256,fp);

          fscanf(fp, "%s %s %lf\n", temp_val_1, temp_val_2,
&parameter[0].step_size);
          fscanf(fp, "%s %s %lf\n", temp_val_1, temp_val_2,
&parameter[0].min_rad);
          fscanf(fp, "%s %s %s\n", temp_val_1, temp_val_2, temp_val_3);

          if (*temp_val_3 == *xy)
            parameter[0].x_y_z = 1;
          else if (*temp_val_3 == *xz)
            parameter[0].x_y_z = 2;
          else if (*temp_val_3 == *yz)
            parameter[0].x_y_z = 3;

          printf("Step size: %lf, Min Rad: %lf, Ana Plane: %d\n",
parameter[0].step_size, parameter[0].min_rad, parameter[0].x_y_z);

          fgets(temp_val_1,256,fp);
          fgets(temp_val_1,256,fp);

          fscanf(fp, "%s %s %s %lf\n", temp_val_1, temp_val_2, temp_val_3,
&parameter[0].centroid[0]);
          fscanf(fp, "%s %s %s %lf\n", temp_val_1, temp_val_2, temp_val_3,
&parameter[0].centroid[1]);

          printf("Centre points: %lf, %lf\n", parameter[0].centroid[0],
parameter[0].centroid[1]);
        }
    };
  printf("Parameters read.\n");
}

/*
*********************************************************************
*/

int read_data(FILE *fp1, FILE *fp2, FILE *fp3, struct node_data *data,
              struct ana_options *parameter)
{
  /*
```

```
  read_data is programmed to read the information contained in two output
  files generated by the optimisation process.

  The first contains the geometry and parameter data for the current
  iteration, the second is the stress results file.

  Information from each file is read into the appropriate structure with
  the stress data being modified initially to return required information
  specific to the type of analysis being performed.

  The data points for the initial shape are also read and stored in the
  data struct.

  */

  struct stress *temp;

  void manipulate_stress(struct node_data *, struct ana_options *, struct stress
*, int, int);
  void calc_fatigue(struct node_data *, struct ana_options *, struct stress *,
int, int);

  int i = 0, j = 0, counter = 0;
  int count_param, node_parameter, random_no, stress_param, temp_store_1;
  double temp_store_2;

  char *fgets();

  /* read geometry data first */

  while(i == 0)
    {
      fscanf(fp3, "%d\n", &temp_store_1);
      for (j = 0; j < parameter[0].num_node; j++)
        {
          fscanf(fp1, "%lf %lf %lf %lf\n", &data[j].x, &data[j].y, &data[j].z,
&data[j].con);
          fscanf(fp3, "%d %lf %lf %lf %lf\n", &temp_store_1, &data[j].x_0,
&data[j].y_0, &data[j].z_0, &temp_store_2);

          data[j].n_x = data[j].x;
          data[j].n_y = data[j].y;
          data[j].n_z = data[j].z;

          if (j == parameter[0].num_node - 1)
            i++;
        };
    }

  printf("Geometry data read.\n");

  /* change num_node to only represent the first layer for a non-constant t
analysis */

  if (parameter[0].options[6] == 1)
    parameter[0].num_node = parameter[0].num_node/2;

  /* read stress data, note that the number of nodes to be read is dependant on
```

```
the
     analysis type */

  /* through thickness analyses require more points initially */

  if (parameter[0].options[6] == 1)
    count_param = 3;
  else if (parameter[0].options[3] == 1)
    count_param = 2;
  else
    count_param = 1;

  if (parameter[0].options[7] == 1)
    node_parameter = parameter[0].num_node * parameter[0].num_loadcase;
  else
    node_parameter = parameter[0].num_node;

  if (count_param > 1)
    stress_param = node_parameter * (count_param + parameter[0].mesh_param[1]);
  else
    stress_param = node_parameter;

  temp = (struct stress *) calloc(2 * stress_param, sizeof(struct stress));

  /* read through white space and header at start of file */

  /* read stress values into temporary array */

  while(i==1)
    {
      for (j = 0; j < stress_param; j++)
        {
          fscanf(fp2, "%d %d %lf %lf\n", &random_no, &counter, &temp[j].s1,
&temp[j].s2);
          if (j == stress_param - 1)
            i++;
        };
    }

  /*
     manipulate stress data depending on analysis type
     for a basic stress analysis the peak stress at each node is assigned as the
modification parameter
     for a quasi 3d stress analysis the peak stress through the thickness (based
on the assigned node ids)
     is used as the modification parameter
     for a fatigue analysis the peak stress is determined at each node for each
loadcase. This is then used for
     a fatigue life calculation, the sum at each node is used as the modification
parameter
  */

  manipulate_stress(data, parameter, temp, node_parameter, count_param);

  free(temp);

  return 0;
}
```

```
/*
*************************************************
*/

void manipulate_stress(struct node_data *data, struct ana_options *parameter,
                       struct stress *temp, int node_param, int count_param)
{
  int i, j, k, start, end, num_loop = 0, node_start, node_end;
  int node_mod, *multipeak;
  double *stress_store, stress_state, threshold, weight_factor;

  stress_store = (double *) calloc(2 * node_param, sizeof(double));

  /* determine peak stress and assign to node modification parameter for each
node*/

  printf("Starting stress manipulations and optimisation parameter
calculations.\n");

  /* consolidate max stress for each combination of node and loadcase */

  for (i = 0; i < node_param; i++)
    {
      stress_store[i] = temp[i].s1;

      if (count_param == 1)
        {
          for(j = 0; j < (parameter[0].mesh_param[1]); j++)
            {
              if(fabs(temp[i + j * node_param].s1) > fabs(temp[i + j *
node_param].s2))
                {
                  if(fabs(temp[i + j * node_param].s1) > stress_store[i])
                    stress_store[i] = temp[i + j * node_param].s1;
                }
              else if(fabs(temp[i + j * node_param].s2) > stress_store[i])
                stress_store[i] = temp[i + j * node_param].s2;
            }
        }
      else
        {
          /* consolidate for quasi 3d through thickness nodes */

          for(j = 0; j < (parameter[0].mesh_param[1] + count_param); j++)
            {
              if ((j == 2) && (count_param == 3)){
                /* do nothing */
              }
              else if ((j==1) && (count_param == 2))
                {
                /* do nothing */
                }
              else {
                if(fabs(temp[i + j * node_param].s1) > fabs(temp[i + j *
node_param].s2))
                  {
                    if(fabs(temp[i + j * node_param].s1) > stress_store[i])
                      stress_store[i] = temp[i + j * node_param].s1;
                  }
```

```
                else if(fabs(temp[i + j * node_param].s2) > stress_store[i])
                  {
                    stress_store[i] = temp[i + j * node_param].s2;
                  }
              }
            }
        }
     }

  for (j = 0; j < parameter[0].num_loadcase; j++)
    {
      /* loop through for each loadcase */

      /* store peak stress in struct */

      for (i = 0; i < parameter[0].num_node; i++)
        {
          node_mod = j + i * parameter[0].num_loadcase;
          if (j > 0)
            {
              if (fabs(stress_store[node_mod]) > fabs(data[i].mp1))
                data[i].mp1 = stress_store[node_mod];
            }
          else
            data[i].mp1 = stress_store[node_mod];
        }
    }

  if ((parameter[0].options[12] == 1)||(parameter[0].options[7] == 0))
    {
      /* assign combined stress back to stress_store */

      for (i = 0; i < parameter[0].num_node; i++)
        stress_store[i] = data[i].mp1;

      /* assign number of loadcase to show only a combined case remains */
      parameter[0].num_loadcase = 1;
    }

  /* calculate crossover stress if multipeak analysis and flag cross over nodes */

  printf("Determining multipeak cross overs.\n");

  for (j = 0; j < parameter[0].num_loadcase; j++)
    {
      /* loop through for each loadcase */

      /* clear current cross_over points */
      if (j > 0)
        {
          num_loop = 0;
          for (i = 0; i < parameter[0].num_node; i++)
            data[i].x_over = 0;
        }

      if(parameter[0].options[0] == 1)
        {
          stress_state = stress_store[j];
          for (i = 0; i < parameter[0].num_node; i++)
```

```
            {
              node_mod = j + i * parameter[0].num_loadcase;
              if (((stress_state > 0.0) && (stress_store[node_mod] <
0.0))||((stress_state < 0.0) && (stress_store[node_mod] > 0.0)))
                {
                  stress_state = stress_store[node_mod];
                  data[i].x_over = 1;
                  num_loop++;
                }
              if (i == parameter[0].num_node - 1)
                {
                  if (((stress_store[node_mod] > 0.0) && (stress_store[j] <
0.0))||((stress_store[node_mod] < 0.0) && (stress_store[j] > 0.0)))
                    data[0].x_over = 1;
                }
            }
          if (num_loop < 1)
            num_loop = 1;
          /* account for open profile (requires extra loop) */
          if (parameter[0].options[17] == 0)
            num_loop++;
        }
      else
        /* 1 loop if single peak analysis */
        num_loop = 1;

      /* calculate stress threshold */
      if (num_loop > 1)
        {
          multipeak = (int *) calloc(4 * num_loop, sizeof(int));
          k = 0;
          for (i = 0; i < parameter[0].num_node; i++)
            {
              node_mod = j + i * parameter[0].num_loadcase;
              if (data[i].x_over == 1)
                {
                  multipeak[k] = i;
                  k++;
                }
              data[i].thresh = stress_store[node_mod];
            }

          for(k = 0; k < num_loop; k++)
            {
              if (parameter[0].options[17] == 1)
                {
                  /* closed profile */
                  if(k == 0)
                    {
                      start = multipeak[num_loop - 1];
                      end = multipeak[k] - 1;
                    }
                  else
                    {
                      start = multipeak[k - 1];
                      end = multipeak[k] - 1;
                    }
                }
              else
```

```
  {
    /* open profile */
    if (k == 0)
      {
        start = 1;
        end = multipeak[k] - 1;
      }
    else if (k == num_loop - 1)
      {
        start = multipeak[k-1];
        end = parameter[0].num_node;
      }
    else
      {
        start = multipeak[k - 1];
        end = multipeak[k] - 1;
      }
  }

/* determine threshold stress for each region */

if (end < start)
  {
    threshold = stress_store[start + 1 + j];
    for (i = start; i < parameter[0].num_node; i++)
      {
        node_mod = j + i * parameter[0].num_loadcase;
        if(fabs(stress_store[node_mod]) > threshold)
          threshold = fabs(stress_store[node_mod]);
      }
    for (i = 0; i < end + 1; i++)
      {
        node_mod = j + i * parameter[0].num_loadcase;
        if(fabs(stress_store[node_mod]) > threshold)
          threshold = fabs(stress_store[node_mod]);
      }
  }
else
  {
    threshold = stress_store[start + 1 + j];
    for(i = start; i < end + 1; i++)
      {
        node_mod = j + i * parameter[0].num_loadcase;
        if(fabs(stress_store[node_mod]) > threshold)
          threshold = fabs(stress_store[node_mod]);
      }
  }

/* assign threshold to each node */
if (end < start)
  {
    for (i = start; i < parameter[0].num_node; i++)
        data[i].thresh = threshold;
    for (i = 0; i < end + 1; i++)
        data[i].thresh = threshold;
  }
else
  {
    for(i = start; i < end + 1; i++)
```

```
                    data[i].thresh = threshold;
                }
            }
        }
      else
        {
          /* calculate threshold stress for single peak*/
          threshold = stress_store[j];
          for (i = 0; i < parameter[0].num_node; i++)
            {
              node_mod = j + i * parameter[0].num_loadcase;
              if (fabs(stress_store[node_mod])> threshold)
                threshold = fabs(stress_store[node_mod]);
            }
          for (i = 0; i < parameter[0].num_node; i++)
            data[i].thresh = threshold;
        }

      /* calculate optimisation parameter d */

      if (parameter[0].options[0] == 1)
        {
          node_start = 0;
          node_end = parameter[0].num_node;
        }
      else
        {
          if (parameter[0].options[17] == 1)
            {
              node_start = 0;
              node_end = parameter[0].num_node;
            }
          else
            {
              node_start = 1;
              node_end = parameter[0].num_node - 1;
              data[0].d = 0;
              data[parameter[0].num_node].d = 0;
            }
        }

      for (i = node_start; i < node_end; i++)
        {
          if (parameter[0].options[13] == 1)
            {
              node_mod = j + i * parameter[0].num_loadcase;
              stress_store[node_mod] = ((fabs(stress_store[node_mod]) -
data[i].thresh)/data[i].thresh) * parameter[0].step_size;
            }
          else if ((parameter[0].options[12] == 1)||(parameter[0].options[7]==0))
            {
              node_mod = j + i * parameter[0].num_loadcase;
              stress_store[node_mod] = ((fabs(data[i].mp1) -
data[i].thresh)/data[i].thresh) * parameter[0].step_size;
            }
        }

      free(multipeak);
    }
```

```
  if ((parameter[0].options[7] == 1)&&(parameter[0].options[13] == 1))
    {
      for (j = 0; j < parameter[0].num_loadcase; j++)
        {
          for (i = node_start; i < node_end; i++)
            {
              node_mod = j + i * parameter[0].num_loadcase;
              if (j==0)
                data[i].d = stress_store[node_mod];
              else if (fabs(stress_store[node_mod]) < fabs(data[i].d))
                data[i].d = stress_store[node_mod];
            }
        }
    }
  else
    {
      for (i = node_start; i < node_end; i++)
        data[i].d = stress_store[i];
    }

  /*
    smooth modification factor at cross over points
    done by averaging the two nodes on either side of the cross over
  */

  if (parameter[0].options[13] == 0 && parameter[0].options[18] == 0 )
    {
      printf("This point shouldn't matter.\n");
      printf("  num_loop=%d\n",num_loop);
      if (num_loop > 1)
        {
          for (j = 0; j < num_loop; j++)
            {
              weight_factor = data[multipeak[j]].d/data[multipeak[j]-1].d;
             printf("  j                     =%d\n",j);
             printf("  weight_factor         =%f\n",weight_factor);
             printf("  multipeak[j]          =%d\n",multipeak[j]);
             printf("  data[multipeak[j]  ].d=%f\n",data[multipeak[j]].d);
             printf("  data[multipeak[j]-1].d=%f\n",data[multipeak[j]-1].d);
              if (weight_factor > 1)
                 weight_factor = 1/weight_factor;
             printf("  weight_factor=%f\n",weight_factor);
              data[multipeak[j]].d = (data[multipeak[j]].d + data[multipeak[j]-
1].d)/2;
              if (data[multipeak[j]-1].d < data[multipeak[j]].d)
               {
                  data[multipeak[j]-1].d = data[multipeak[j]].d;
                  data[multipeak[j]  ].d = data[multipeak[j]].d * weight_factor;
                }
              else
                data[multipeak[j]-1].d = data[multipeak[j]].d * weight_factor;
              weight_factor = data[multipeak[j]+1].d/data[multipeak[j]-2].d;
              if (weight_factor > 1)
                weight_factor = 1/weight_factor;
              data[multipeak[j]+1].d = (data[multipeak[j]+1].d +
data[multipeak[j]-2].d)/2;
              if (data[multipeak[j]-2].d < data[multipeak[j]+1].d)
                {
```

```
                data[multipeak[j]-2].d = data[multipeak[j]+1].d;
                data[multipeak[j]+1].d = data[multipeak[j]+1].d * weight_factor;
              }
            else
              data[multipeak[j]-2].d = data[multipeak[j]+1].d * weight_factor;
          }
      }
  }
  printf("Finished manipulating stress and determining optimisation factors.\n");
  free(stress_store);
}


/*
*********************************************************************
*/

void calc_fatigue(struct node_data *data, struct ana_options *parameter,
                  struct stress *temp, int node_param, int count_param)
{
  int i, j, k;
  double load_dist[parameter[0].num_loadcase];
  double Su, S3, S6, life, a, b;

  /*
     Determine peak stress and assign to temporary storage (first elements of temp
array).
     Only positive stresses contribute to the fatigue at this stage so only +ve
stresses stored.
  */

  for (i = 0; i < node_param; i++)
    {
      if (count_param == 1)
        {
            {
              if(temp[i].s1 < temp[i].s2)
                temp[i].s1 = temp[i].s2;
            }
        }
      else
        {
          for(j = 0; j <= (parameter[0].mesh_param[2] + count_param); j++)
            {
              if((temp[i].s1 < temp[i].s2)&&(j==0))
                temp[i].s1 = temp[i].s2;
              else if((temp[i].s1 < temp[i + j * node_param].s1)||(temp[i].s1 <
temp[i + j * node_param].s2))
                {
                  if (temp[i + j * node_param].s1 < temp[i + j * node_param].s2)
                    temp[i].s1 = temp[i + j * node_param].s2;
                  else
                    temp[i].s1 = temp[i + j * node_param].s1;
                }
            }
        }
      if (temp[i].s1 < 0)
        temp[i].s1 = 0;
    }
```

```
  /* once stresses for each load case are determined calculate fatigue life index
for each node based on loadcase distribution */

  /* determine distribution constant, normal or .... */
  if (parameter[0].options[19] == 1)
    {
      for (i = 0; i < parameter[0].num_loadcase; i++)
        {
          load_dist[i] = 1.0/parameter[0].num_loadcase;
        }
    }
  else if (parameter[0].options[19] == 0)
    {
      for (i = 0; i < parameter[0].num_loadcase; i++)
        load_dist[i] = 1/sqrt(2*3.141)*exp(-0.5*(i-1-
pow((parameter[0].num_loadcase-1)/2, 2)));
    }

  /* calculate fatigue damage rate based on material S-N curve */

  Su = 60000;
  S3 = 0.9 * Su;
  S6 = 0.5 * Su;
  a = (log10(S3) - log10(S6))/(log10(1000)-log10(1000000));
  b = log10(S3) - a * log10(1000);

  k = 0;

  for (i = 0; i < node_param; i = i + parameter[0].num_loadcase)
    {
      data[k].mp1 = 0;
      for (j = 0; j < parameter[0].num_loadcase; j++)
        {
          if (temp[i + j].s1 > S6)
            {
              life = pow(10, (log10(temp[i + j].s1) - b)/a);
              data[k].mp1 = data[k].mp1 + load_dist[j]/life;
            }
        }
      k++;
    }
}

/*
*********************************************************************
*/

void extract_xyz(struct ana_options *parameter, double *xyz, int i, struct
node_data *data)
{
  /* extracts the relevant values of x, y or z depending on the analysis type and
plane */

  if(parameter[0].options[17] == 1)
    {
      if (parameter[0].x_y_z == 1)
        {
          if (i == 0)
```

```
      {
        xyz[0] = data[parameter[0].num_node - 1].x;
        xyz[1] = data[i].x;
        xyz[2] = data[i + 1].x;
        xyz[3] = data[parameter[0].num_node - 1].y;
        xyz[4] = data[i].y;
        xyz[5] = data[i + 1].y;
      }
    else if(i == parameter[0].num_node - 1)
      {
        xyz[0] = data[i - 1].x;
        xyz[1] = data[i].x;
        xyz[2] = data[0].x;
        xyz[3] = data[i - 1].y;
        xyz[4] = data[i].y;
        xyz[5] = data[0].y;
      }
    else
      {
        xyz[0] = data[i - 1].x;
        xyz[1] = data[i].x;
        xyz[2] = data[i + 1].x;
        xyz[3] = data[i - 1].y;
        xyz[4] = data[i].y;
        xyz[5] = data[i + 1].y;
      };
  }
else if (parameter[0].x_y_z == 2)
  {
    if (i == 0)
      {
        xyz[0] = data[parameter[0].num_node - 1].x;
        xyz[1] = data[i].x;
        xyz[2] = data[i + 1].x;
        xyz[3] = data[parameter[0].num_node - 1].z;
        xyz[4] = data[i].z;
        xyz[5] = data[i + 1].z;
      }
    else if(i == parameter[0].num_node - 1)
      {
        xyz[0] = data[i - 1].x;
        xyz[1] = data[i].x;
        xyz[2] = data[0].x;
        xyz[3] = data[i - 1].z;
        xyz[4] = data[i].z;
        xyz[5] = data[0].z;
      }
    else
      {
        xyz[0] = data[i - 1].x;
        xyz[1] = data[i].x;
        xyz[2] = data[i + 1].x;
        xyz[3] = data[i - 1].z;
        xyz[4] = data[i].z;
        xyz[5] = data[i + 1].z;
      };
  }
else if (parameter[0].x_y_z == 3)
  {
```

```
            if (i == 0)
              {
                xyz[0] = data[parameter[0].num_node - 1].y;
                xyz[1] = data[i].y;
                xyz[2] = data[i + 1].y;
                xyz[3] = data[parameter[0].num_node - 1].z;
                xyz[4] = data[i].z;
                xyz[5] = data[i + 1].z;
              }
            else if(i == parameter[0].num_node - 1)
              {
                xyz[0] = data[i - 1].y;
                xyz[1] = data[i].y;
                xyz[2] = data[0].y;
                xyz[3] = data[i - 1].z;
                xyz[4] = data[i].z;
                xyz[5] = data[0].z;
              }
            else
              {
                xyz[0] = data[i - 1].y;
                xyz[1] = data[i].y;
                xyz[2] = data[i + 1].y;
                xyz[3] = data[i - 1].z;
                xyz[4] = data[i].z;
                xyz[5] = data[i + 1].z;
              };
          }
      }
    else
      {
        if (parameter[0].x_y_z == 1)
          {
            xyz[0] = data[i - 1].x;
            xyz[1] = data[i].x;
            xyz[2] = data[i + 1].x;
            xyz[3] = data[i - 1].y;
            xyz[4] = data[i].y;
            xyz[5] = data[i + 1].y;
          }
        else if (parameter[0].x_y_z == 2)
          {
            xyz[0] = data[i - 1].x;
            xyz[1] = data[i].x;
            xyz[2] = data[i + 1].x;
            xyz[3] = data[i - 1].z;
            xyz[4] = data[i].z;
            xyz[5] = data[i + 1].z;
          }
        else if (parameter[0].x_y_z == 3)
          {
            xyz[0] = data[i - 1].y;
            xyz[1] = data[i].y;
            xyz[2] = data[i + 1].y;
            xyz[3] = data[i - 1].z;
            xyz[4] = data[i].z;
            xyz[5] = data[i + 1].z;
          }
      }
```

```
}

/*
**********************************************************************
*/

void calculate_normal(struct ana_options *parameter, double *xyz,
                      int i, struct node_data *data)
{
  double diff1, diff2, result_angle;

  diff1 = xyz[2] - xyz[0];
  diff2 = xyz[5] - xyz[3];

  result_angle = atan2(diff2, diff1);
  data[i].normal = (result_angle * angle_mod + 90.0);

  if(data[i].normal > 180)
    data[i].normal = data[i].normal - 360;

  if ((parameter[0].options[2] == 0) && (data[i].normal < 0))
    data[i].normal = data[i].normal + 180;
  else if ((parameter[0].options[2] == 0) && (data[i].normal > 0))
    data[i].normal = data[i].normal - 180;
}

/*
**********************************************************************
*/

void do_node_move(struct node_data *data, struct ana_options *parameter, int i)
{
  int clock_dir = -1;

  /* perform movement to new coords */

  printf("node = %4d, d = %12.6f, normal = %.6f\n", i, data[i].d, data[i].normal);

  if(data[i].con == 0.0)
    { /* free in both directions */
      if(parameter[0].x_y_z == 1.0)
        {
          data[i].x = data[i].x + data[i].d*cos(data[i].normal / angle_mod) *
clock_dir;
          data[i].y = data[i].y + data[i].d*sin(data[i].normal / angle_mod) *
clock_dir;
        }
      else if(parameter[0].x_y_z == 2.0)
        {
          data[i].x = data[i].x + data[i].d* cos(data[i].normal / angle_mod) *
clock_dir;
          data[i].z = data[i].z + data[i].d* sin(data[i].normal / angle_mod) *
clock_dir;
        }
      else if(parameter[0].x_y_z == 3.0)
        {
          data[i].y = data[i].y + data[i].d * cos(data[i].normal / angle_mod) *
clock_dir;
          data[i].z = data[i].z + data[i].d * sin(data[i].normal / angle_mod) *
```

```
clock_dir;
        }
    }
  else if(data[i].con == 1.0)
    { /* fixed in second dimension */
      if(parameter[0].x_y_z == 1.0)
        {
          data[i].y = data[i].y + data[i].d * sin(data[i].normal / angle_mod) *
clock_dir;
        }
      else if(parameter[0].x_y_z == 2.0)
        {
          data[i].z = data[i].z + data[i].d * sin(data[i].normal / angle_mod) *
clock_dir;
        }
      else if(parameter[0].x_y_z == 3.0)
        {
          data[i].z = data[i].z + data[i].d * sin(data[i].normal / angle_mod) *
clock_dir;
        }
    }
  else if(data[i].con == 2.0)
    { /* fixed in first dimension */
      if(parameter[0].x_y_z == 1.0)
        {
          data[i].x = data[i].x + data[i].d * cos(data[i].normal / angle_mod) *
clock_dir;
        }
      else if(parameter[0].x_y_z == 2.0)
        {
          data[i].x = data[i].x + data[i].d * cos(data[i].normal / angle_mod) *
clock_dir;
        }
      else if(parameter[0].x_y_z == 3.0)
        {
          data[i].y = data[i].y + data[i].d * cos(data[i].normal / angle_mod) *
clock_dir;
        }
    }
  else if(data[i].con == 3.0)
    { /* fixed in both directions */
      if(parameter[0].x_y_z == 1.0)
        {
          data[i].x = data[i].x;
          data[i].y = data[i].y;
        }
      else if(parameter[0].x_y_z == 2.0)
        {
          data[i].x = data[i].x;
          data[i].z = data[i].z;
        }
      else if(parameter[0].x_y_z == 3.0)
        {
          data[i].y = data[i].y;
          data[i].z = data[i].z;
        }
    }
}
```

```
/*
**********************************************************************
*/

void do_optimise_move(struct node_data *data, struct ana_options *parameter)
{

  /* performs nodal movement based on parameters determined in optimisation
calculations */

  int i, node_start, node_end;
  double xyz[6], dd, ff;

  if (parameter[0].options[17] == 1)
    {
      node_start = 0;
      node_end = parameter[0].num_node;
    }
  else
    {
      node_start = 1;
      node_end = parameter[0].num_node - 1;
    }

  printf("Node start = %d\n", node_start);
  printf("Node end   = %d\n", node_end-1);

  for (i = node_start; i < node_end; i++)
    {
      /* calculate normal direction for each node (based on 3 point analysis) */

      extract_xyz(parameter, xyz, i, data);

      calculate_normal(parameter, xyz, i, data);

      /* calculations necessary for space ratio */

      dd = sqrt(pow((xyz[2] - xyz[0]),2) + pow((xyz[5] - xyz[3]),2));
      ff = sqrt(pow((xyz[1] - xyz[0]),2) + pow((xyz[4] - xyz[3]),2));

      /* node_space_ratio */

      data[i].space_r = ff/dd;
    }

  for (i = node_start; i < node_end; i++)
    {
      /* perform nodal movement */
      do_node_move(data, parameter, i);
    }
  printf("Finished node movement.\n");
}

/*
**********************************************************************
*/

void apply_bound_constraint(struct ana_options *parameter, struct node_data *data)
{
```

```
/* check boundary constraints and move nodes if outside the bounding box */

int i, node_start, node_end;

if (parameter[0].options[17] == 1)
  {
    node_start = 0;
    node_end = parameter[0].num_node;
  }
else
  {
    node_start = 1;
    node_end = parameter[0].num_node - 1;
  }

if(parameter[0].x_y_z == 1.0)
  {
    for(i = node_start; i < node_end; i++)
      {
        if(data[i].x < parameter[0].constraints[0])
          {
            data[i].x = parameter[0].constraints[0];
            printf("Boundary x constraint enforced node: %d\n", i+1);
          }
        else if (data[i].x > parameter[0].constraints[1])
          {
            data[i].x = parameter[0].constraints[1];
            printf("Boundary x constraint enforced node: %d\n", i+1);
          }
        if(data[i].y < parameter[0].constraints[2])
          {
            data[i].y = parameter[0].constraints[2];
            printf("Boundary y constraint enforced node: %d\n", i+1);
          }
        else if(data[i].y > parameter[0].constraints[3])
          {
            data[i].y = parameter[0].constraints[3];
            printf("Boundary y constraint enforced node: %d\n", i+1);
          }
      }
  }
else if(parameter[0].x_y_z == 2.0)
  {
    for(i = node_start; i < node_end; i++)
      {
        if(data[i].x < parameter[0].constraints[0])
          data[i].x = parameter[0].constraints[0];
        else if (data[i].x > parameter[0].constraints[1])
          data[i].x = parameter[0].constraints[1];

        if(data[i].z < parameter[0].constraints[2])
          data[i].z = parameter[0].constraints[2];
        else if(data[i].z > parameter[0].constraints[3])
          data[i].z = parameter[0].constraints[3];
      }
  }
else if(parameter[0].x_y_z == 3.0)
  {
    for(i = node_start; i < node_end; i++)
```

```
        {
          if(data[i].y < parameter[0].constraints[0])
            data[i].y = parameter[0].constraints[0];
          else if (data[i].y > parameter[0].constraints[1])
            data[i].y = parameter[0].constraints[1];

          if(data[i].z < parameter[0].constraints[2])
            data[i].z = parameter[0].constraints[2];
          else if(data[i].z > parameter[0].constraints[3])
            data[i].z = parameter[0].constraints[3];
        }
    }
}

/*
**********************************************************************
*/

void output_results(struct node_data *data, struct ana_options *parameter)
{
  /* output results of data analysis to file */

  FILE *output_file;
  FILE *sum_file;
  int j;

  output_file = fopen("nodes.opt","w");

  fprintf(output_file, "%d\n",parameter[0].num_node);

  for (j = 0; j < parameter[0].num_node; j++)
    {
      fprintf(output_file, "%6d %13.6lf %13.6lf %13.6lf %13.6lf\n", j+1,
               data[j].x, data[j].y, data[j].z, data[j].con);
    };

  fclose(output_file);

  sum_file = fopen("summary_file.dat","w");

  for (j = 0; j < parameter[0].num_node; j++)
    {
      fprintf(sum_file, "%5d %13.6lf %13.6lf %13.6lf %13.6lf %13.6lf %16.6lf
%13.6lf %13.6lf %13.6lf\n",
              j+1,
               data[j].x, data[j].y, data[j].z, data[j].space_r, data[j].normal,
               data[j].rad, data[j].d, data[j].mp1, data[j].thresh);
    };

  fclose(sum_file);
}

/*
**********************************************************************
*/

void do_bound_mods(struct node_data *data, struct ana_options *parameter)
{
  void apply_bound_constraint(struct ana_options *, struct node_data *);
```

```
  void rad_calc(struct ana_options *, double *, int, struct node_data *);
  void rad_mod_check(struct node_data *data, int i, struct ana_options
*parameter);
  void modify_angle(int, struct node_data *, double *, struct ana_options *,
double *);

  int i, j, iterate = 1, count = 1, det_flag, node_start, node_end, mod_done;
  double xyz[6];
  double alpha[1], gamma[1], theta[1], dd, ee, ff;
  double current_min, con_factor;

  if (parameter[0].options[17] == 1)
    {
      node_start = 0;
      node_end = parameter[0].num_node;
    }
  else
    {
      node_start = 1;
      node_end = parameter[0].num_node - 1;
    }

  while (iterate == 1)
    {
      /* check bounding box first and adjust nodes accordingly */

      apply_bound_constraint(parameter, data);

      /* perform minimum radius calculations */

      current_min = parameter[0].min_rad;

      for(i = node_start; i < node_end; i++)
        {
          rad_calc(parameter, xyz, i, data);
        }

      /* check and adjust radius */

      for(i = node_start; i < node_end; i++)
        {
          extract_xyz(parameter, xyz, i, data);
          calculate_normal(parameter, xyz, i, data);

          dd = sqrt(pow((xyz[2] - xyz[0]),2) + pow((xyz[5] - xyz[3]),2));

          if (parameter[0].options[2] == 1)
            ee = sqrt((pow(parameter[0].min_rad,2)) - (pow((dd/2.0),2)));
          else
            ee = sqrt((pow(parameter[0].min_rad,2)) - (pow((dd/2.0),2)));

          if ((data[i].rad == 0.0)||(parameter[0].min_rad == 0.0))
            {
              /* if convex or on straight line move to straight line (determinant
<= 0.0) */
              if (parameter[0].x_y_z == 1)
                {
                  if (data[i].con == 0)
                    {
```

```
                    data[i].x = xyz[0] * (1 - data[i].space_r) + xyz[2] *
data[i].space_r;
                    data[i].y = xyz[3] * (1 - data[i].space_r) + xyz[5] *
data[i].space_r;
                  }
                else if (data[i].con == 1)
                  {
                    data[i].y = (data[i].y + xyz[3] * (1 - data[i].space_r) +
xyz[5] * data[i].space_r)/2;
                  }
                else if (data[i].con == 2)
                  {
                    data[i].x = (data[i].x + xyz[2] * (1 - data[i].space_r) +
xyz[0] * data[i].space_r)/2;
                  }
              }
            else if (parameter[0].x_y_z == 2)
              {
                if (data[i].con == 0)
                  {
                    data[i].x = xyz[0] * (1 - data[i].space_r) + xyz[2] *
data[i].space_r;
                    data[i].z = xyz[3] * (1 - data[i].space_r) + xyz[5] *
data[i].space_r;
                  }
                else if (data[i].con == 1)
                  {
                    data[i].z = (data[i].z + xyz[3] * (1 - data[i].space_r) +
xyz[5] * data[i].space_r)/2;
                  }
                else if (data[i].con == 2)
                  {
                    data[i].x = (data[i].x + xyz[0] * (1 - data[i].space_r) +
xyz[2] * data[i].space_r)/2;
                  }
              }
            else if (parameter[0].x_y_z == 3)
              {
                if (data[i].con == 0)
                  {
                    data[i].y = xyz[0] * (1 - data[i].space_r) + xyz[2] *
data[i].space_r;
                    data[i].z = xyz[3] * (1 - data[i].space_r) + xyz[5] *
data[i].space_r;
                  }
                else if (data[i].con == 1)
                  {
                    data[i].z = (data[i].z + xyz[3] * (1 - data[i].space_r) +
xyz[5] * data[i].space_r)/2;
                  }
                else if (data[i].con == 2)
                  {
                    data[i].y = (data[i].y + xyz[0] * (1 - data[i].space_r) +
xyz[2] * data[i].space_r)/2;
                  }
              }
          }
        else if (data[i].rad < parameter[0].min_rad)
          {
```

```
                /*
                   ron's method
                   determine position of point necessary to meet min radius
                   and adjust position accordingly
                */

                if ((i == 0)||(i == 36))
                  printf("node: %d, rad: %lf, x1: %lf, y1: %lf, x2: %lf, y2: %lf,
x3: %lf, y3: %lf\n", i + 1, data[i].rad, xyz[0], xyz[3], xyz[1], xyz[4], xyz[2],
xyz[5]);

                theta[0] = atan((parameter[0].min_rad-ee)/(dd/2));

                ff = sqrt(pow((parameter[0].min_rad - ee),2) + pow((dd/2),2));
                alpha[0] = atan((xyz[5] - xyz[3])/(xyz[2]-xyz[0]));

                if (parameter[0].options[2] == 1)
                  gamma[0] = (alpha[0] + theta[0]) * angle_mod;
                else
                  gamma[0] = (alpha[0] - theta[0]) * angle_mod;

                /* modify application angle (gamma[0]) based on normals) */

                modify_angle(i, data, gamma, parameter, xyz);

                if (data[i].con > 0.0)
                  {
                    if ((data[i].space_r > 0.51)||(data[i].space_r < 0.49))
                      {
                        if (data[i].con == 1.0)
                          con_factor = (xyz[0]-xyz[1])/(ff*cos(gamma[0]/angle_mod));
                        else if (data[i].con == 2.0)
                          con_factor = (xyz[3]-xyz[4])/(ff*sin(gamma[0]/angle_mod));
                        if (parameter[0].options[2] == 0.0)
                          con_factor = -con_factor;
                        printf("node id: %d, con factor: %lf\n", i + 1, con_factor);
                      }
                  }
                else
                  con_factor = 1.0;

                if (data[i].con == 0.0)
                  {
                    if (parameter[0].x_y_z == 1)
                      {
                        data[i].x = xyz[0] + ff * cos(gamma[0] / angle_mod);
                        data[i].y = xyz[3] + ff * sin(gamma[0] / angle_mod);
                      }
                    else if (parameter[0].x_y_z == 2)
                      {
                        data[i].x = xyz[0] + ff * cos(gamma[0] / angle_mod);
                        data[i].z = xyz[3] + ff * sin(gamma[0] / angle_mod);
                      }
                    else if (parameter[0].x_y_z == 3)
                      {
                        data[i].y = xyz[0] + ff * cos(gamma[0] / angle_mod);
                        data[i].z = xyz[3] + ff * sin(gamma[0] / angle_mod);
                      }
                  }
```

```
              else if (data[i].con == 1.0)
                {
                  if ((parameter[0].x_y_z == 3)||(parameter[0].x_y_z == 2))
                    {
                      data[i].z = xyz[3] + ff * con_factor * sin(gamma[0] /
angle_mod);
                    }
                  else if (parameter[0].x_y_z == 1)
                    {
                      data[i].y = xyz[3] + ff * con_factor * sin(gamma[0] /
angle_mod);
                    }
                }
              else if (data[i].con == 2.0)
                {
                  if (parameter[0].x_y_z == 3)
                    {
                      data[i].y = xyz[0] + ff * con_factor * cos(gamma[0] /
angle_mod);
                    }
                  else if ((parameter[0].x_y_z == 1)||(parameter[0].x_y_z == 2))
                    {
                      data[i].x = xyz[0] + ff * con_factor * cos(gamma[0] /
angle_mod);
                    }
                }
              if ((i == 0)||(i == 36))
                printf("node: %d, rad: %lf, new x: %lf, new y: %lf\n", i + 1,
data[i].rad, data[i].x, data[i].y);

              mod_done = 1;
            }

          calculate_normal(parameter, xyz, i, data);

          /* ensure new shape does not cross boundary of initial profile. */
          if (parameter[0].options[11] == 1)
            rad_mod_check(data, i, parameter);

          /*
             update radius after node movement
          */
          if (i == node_end)
            {
              rad_calc(parameter, xyz, i, data);
              if (parameter[0].options[17] == 1)
                rad_calc(parameter, xyz, node_start, data);
            }
          else
            {
              rad_calc(parameter, xyz, i, data);
              rad_calc(parameter, xyz, i + 1, data);
            }

          if ((i == 0)||(i == 36))
            printf("node: %d, rad: %lf, new x am: %lf, new y am: %lf, gamma: %lf,
ff: %lf\n", i + 1, data[i].rad, data[i].x, data[i].y, gamma[0], ff);

          if (mod_done == 1)
```

```
            {
              mod_done = 0;
            }
        }

      /* check for minimum radius constraints */

      for (j = node_start; j < node_end; j++)
        {
          rad_calc(parameter, xyz, j, data);

          if ((data[j].rad < current_min) && (current_min != 0.0) && (data[j].rad
!= 0.0))
            current_min = data[j].rad;
          else if (current_min == 0.0)
            current_min = parameter[0].min_rad;

          if (((parameter[0].options[2] == 0) && (data[i].det <
0))||((parameter[0].options[2] == 1) && (data[i].det > 0)))
            det_flag = 1;
        }

      if (count > 100)
        iterate = 0;
      else if (current_min >= parameter[0].min_rad)
        iterate = 0;
      else
        {
          det_flag = 0;
          count++;
        }
    }

  printf("%d radius iterations were performed.\n", count);
}

/*
*********************************************************************
*/

void rad_mod_check(struct node_data *data, int i, struct ana_options *parameter)
{
  /* check that the radius modification constraint has not moved inside original
shape */

  int node_modified = 0;
  double dir_1_diff, dir_2_diff, current_dis, previous_dis;

  if (parameter[0].x_y_z == 1)
    {
      dir_1_diff = data[i].x - data[i].n_x;
      dir_2_diff = data[i].y - data[i].n_y;
      current_dis = sqrt(pow(data[i].x - parameter[0].centroid[0], 2) +
pow(data[i].y - parameter[0].centroid[1], 2));
      previous_dis = sqrt(pow(data[i].n_x - parameter[0].centroid[0], 2) +
pow(data[i].n_y - parameter[0].centroid[1], 2));
    }
  else if (parameter[0].x_y_z == 2)
    {
```

```
      dir_1_diff = data[i].x - data[i].n_x;
      dir_2_diff = data[i].z - data[i].n_z;
      current_dis = sqrt(pow(data[i].x - parameter[0].centroid[0], 2) +
pow(data[i].z - parameter[0].centroid[1], 2));
      previous_dis = sqrt(pow(data[i].n_x - parameter[0].centroid[0], 2) +
pow(data[i].n_z - parameter[0].centroid[1], 2));
    }
  else if (parameter[0].x_y_z == 3)
    {
      dir_1_diff = data[i].y - data[i].n_y;
      dir_2_diff = data[i].z - data[i].n_z;
      current_dis = sqrt(pow(data[i].y - parameter[0].centroid[0], 2) +
pow(data[i].z - parameter[0].centroid[1], 2));
      previous_dis = sqrt(pow(data[i].n_y - parameter[0].centroid[0], 2) +
pow(data[i].n_z - parameter[0].centroid[1], 2));
    }

  if ((data[i].normal >= -180) && (data[i].normal < -90))
    {
      if ((dir_1_diff > 0.0001)||(dir_2_diff > 0.0001))
        {
          if (current_dis < previous_dis)
            {
              if (parameter[0].x_y_z == 1)
                {
                  data[i].x = data[i].n_x;
                  data[i].y = data[i].n_y;
                }
              if (parameter[0].x_y_z == 2)
                {
                  data[i].x = data[i].n_x;
                  data[i].z = data[i].n_z;
                }
              if (parameter[0].x_y_z == 3)
                {
                   data[i].y = data[i].n_y;
                  data[i].z = data[i].n_z;
                }
              node_modified = 2;
            }
        }
    }
  else if ((data[i].normal >= -90) && (data[i].normal < 0))
    {
      if ((dir_1_diff < -0.0001)||(dir_2_diff > 0.0001))
        {
          if (current_dis < previous_dis)
            {
              if (parameter[0].x_y_z == 1)
                {
                  data[i].x = data[i].n_x;
                  data[i].y = data[i].n_y;
                }
              if (parameter[0].x_y_z == 2)
                {
                  data[i].x = data[i].n_x;
                  data[i].z = data[i].n_z;
                }
              if (parameter[0].x_y_z == 3)
```

```
                {
                  data[i].y = data[i].n_y;
                  data[i].z = data[i].n_z;
                }
              node_modified = 2;
            }
        }
    }
  else if ((data[i].normal >= 0) && (data[i].normal < 90))
    {
      if ((dir_1_diff < -0.0001)||(dir_2_diff < -0.0001))
        {
          if (current_dis < previous_dis)
            {
              if (parameter[0].x_y_z == 1)
                {
                  data[i].x = data[i].n_x;
                  data[i].y = data[i].n_y;
                }
              if (parameter[0].x_y_z == 2)
                {
                  data[i].x = data[i].n_x;
                  data[i].z = data[i].n_z;
                }
              if (parameter[0].x_y_z == 3)
                {
                  data[i].y = data[i].n_y;
                  data[i].z = data[i].n_z;
                }
              node_modified = 2;
            }
        }
    }
  else if ((data[i].normal >= 90) && (data[i].normal < 180))
    {
      if ((dir_1_diff > 0.0001)||(dir_2_diff < -0.0001))
        {
          if (current_dis < previous_dis)
            {
              if (parameter[0].x_y_z == 1)
                {
                  data[i].x = data[i].n_x;
                  data[i].y = data[i].n_y;
                }
              if (parameter[0].x_y_z == 2)
                {
                  data[i].x = data[i].n_x;
                  data[i].z = data[i].n_z;
                }
              if (parameter[0].x_y_z == 3)
                {
                  data[i].y = data[i].n_y;
                  data[i].z = data[i].n_z;
                }
              node_modified = 2;
            }
        }
    }
}
```

```
/*
**********************************************************************
*/

void rad_calc(struct ana_options *parameter, double *xyz, int i, struct node_data
*data)
{
  double a1, a2, b1, b2, c1, c2, xc, yc, a12, a22;

  extract_xyz(parameter, xyz, i, data);

  /* calculate determinant */

  a1 = 2 * (xyz[1]-xyz[0]);
  a2 = 2 * (xyz[2]-xyz[1]);
  b1 = 2 * (xyz[4]-xyz[3]);
  b2 = 2 * (xyz[5]-xyz[4]);
  c1 = pow(xyz[1], 2) + pow(xyz[4], 2) - pow(xyz[0], 2) - pow(xyz[3], 2);
  c2 = pow(xyz[2], 2) + pow(xyz[5], 2) - pow(xyz[1], 2) - pow(xyz[4], 2);

  data[i].det = a1*b2-b1*a2;

  if (fabs(data[i].det) < 0.0000001)
    data[i].det = 0;

  /* calculate radius */

  if (((parameter[0].options[2] == 0) && (data[i].det >
0))||((parameter[0].options[2] == 1) && (data[i].det < 0)))
    {
      xc = (c1 * b2 - b1 * c2) / data[i].det;
      yc = (a1 * c2 - c1 * a2) / data[i].det;

      a12 = xc - xyz[0];
      a22 = yc - xyz[3];

      data[i].rad = sqrt(pow(a12,2) + pow(a22,2));
    }
  else
    data[i].rad = 0.0;
}

/*
**********************************************************************
*/

void modify_angle(int i, struct node_data *data, double *angle,
                  struct ana_options *parameter, double *xyz)
{
  /* modifies application angles in radius calcs to match by quadrant,
     option 1 indicates a segment angle modification, options 2
     indicates a gamma angle modification
     uses the pointer to the required angle
  */

  int mod_param;

  calculate_normal(parameter, xyz, i, data);
```

```
  if (parameter[0].options[2] == 1)
    mod_param = 180;
  else if (parameter[0].options[2] == 0)
    mod_param = 0;

  if ((data[i].normal >= -180) && (data[i].normal <= -90))
    {
      if ((angle[0] >= -180) && (angle[0] < -90))
        angle[0] = angle[0] + 90 + mod_param;
      else if ((angle[0] >= -90) && (angle[0] < 0))
        angle[0] = angle[0] + mod_param;
      else if ((angle[0] >= 0) && (angle[0] < 90))
        angle[0] = angle[0] - 90 + mod_param;
      else if ((angle[0] >= 90) && (angle[0] < 180))
        angle[0] = angle[0] - 180 + mod_param;
    }
  else if ((data[i].normal >= -90) && (data[i].normal <= 0))
    {
      if ((angle[0] >= -180) && (angle[0] < -90))
        angle[0] = angle[0] + 180 - mod_param;
      else if ((angle[0] >= -90) && (angle[0] < 0))
        angle[0] = angle[0] + 90 - mod_param;
      else if ((angle[0] >= 0) && (angle[0] < 90))
        angle[0] = angle[0] - mod_param;
      else if ((angle[0] >= 90) && (angle[0] < 180))
        angle[0] = angle[0] - 90 - mod_param;
    }
  else if ((data[i].normal >= 0) && (data[i].normal <= 90))
    {
      if ((angle[0] >= -180) && (angle[0] < -90))
        angle[0] = angle[0] + 270 - mod_param;
      else if ((angle[0] >= -90) && (angle[0] < 0))
        angle[0] = angle[0] + 180 - mod_param;
      else if ((angle[0] >= 0) && (angle[0] < 90))
        angle[0] = angle[0] + 90 - mod_param;
      else if ((angle[0] >= 90) && (angle[0] < 180))
        angle[0] = angle[0] - mod_param;
    }
  else if ((data[i].normal >= 90) && (data[i].normal <= 180))
    {
      if ((angle[0] >= -180) && (angle[0] < -90))
        angle[0] = angle[0] + mod_param;
      else if ((angle[0] >= -90) && (angle[0] < 0))
        angle[0] = angle[0] - 90 + mod_param;
      else if ((angle[0] >= 0) && (angle[0] < 90))
        angle[0] = angle[0] - 180 + mod_param;
      else if ((angle[0] >= 90) && (angle[0] < 180))
        angle[0] = angle[0] - 270 + mod_param;
    }
}

/*
**********************************************************************
*/

void modify_z(struct node_data *data, struct ana_options *parameter)
{
```

```
  int i;
  double hole_rad, flange_rad, flange_dist, y_mod, root_val;

  printf("Starting layer 2 modification.\n");

  /* re-initialise num_node for 2 layers */
  parameter[0].num_node = parameter[0].num_node * 2;

  /* calculate z value for second layer */

  /* defined constants for analysis */

  hole_rad = 3.937;
  flange_rad = 1.524;
  flange_dist = hole_rad + 0.508 - flange_rad;
  y_mod = 6.562446;

  /*
     calculate values, formula supplied by R. Evans for P3 analysis
     algorithm checks if z needs to be modified, ie lies on flange and
     adjusts accordingly
  */

  for (i = parameter[0].num_node/2; i < parameter[0].num_node; i++)
    {
      if ((data[i].y - y_mod <= -flange_dist) & (data[i].y - y_mod >= -(hole_rad +
0.508)))
        {
          root_val = pow(flange_rad, 2) - pow(data[i].y - y_mod + flange_dist, 2);
          data[i].z = parameter[0].mesh_param[0] + flange_rad - sqrt(root_val);
        }
      else
        {
          data[i].z = parameter[0].mesh_param[0];
        }
    }
}

/*
**********************************************************************
*/
```

Shell script optscriptq.sh

```
#!/bin/sh

echo ""
echo "=================================================="
echo "  SHELL SCRIPT FOR RUNNING SHAPE OPTIMISATION JOBS"
echo "=================================================="

time0=$(date +"%T")" "$(date +"%Y-%m-%d")
secs0=$(date +"%s")

echo ""
echo "Job start time: $time0"

# Run the installed Intel-supplied ifortvar.sh shell script
# to create the required environment variables for running the
```

```
# Intel Fortran compiler. The location of the shell script will
# depend on the specific version of the compiler.
#
# Note that these environment variables will remain local to
# the present shell as a result of using the "source" command.

echo ""
echo "Setting up Intel Fortran compiler environment to enable usage"
echo "of an Abaqus user subroutine with the shape optimisation job."

source /opt/intel/Compiler/11.1/069/bin/ifortvars.sh intel64

# Configure the files for performing the shape optimisation,
# as well as performing a cleanup.

cp nodes.ini nodes.dat
cp start.inp opjob.inp

rm -f convergence.dat
rm -f znodes???.dat
rm -f zstress???.dat

# Perform the required number of iterations of the shape
# optimisation code.

maxitns=1

for (( i = 1;  i <= maxitns; i++ ))
do
  echo ""
  echo "====================================="
  echo "Starting iteration $i of $maxitns"
  echo "====================================="
  echo ""
  time1=$(date +"%T")" "$(date +"%Y-%m-%d")
  secs1=$(date +"%s")
  mv opjob.inp temp.inp
  rm -f opjob.*
  mv temp.inp opjob.inp
  abaqus job=opjob user=getsigq cpus=1 interactive
  ./fsig    # Formats and collates stress???.dat files into stress.rpt
  ./rednf   # Reduces format of nodes.dat and puts in nodes.opt for optim4
  ./optim4  # Ron Braemar's C code. No mods have been done that affect function
  ./expnf   # Expands format of nodes.opt and puts into nodes.dat
  cp opjob.inp opjob.temp
  ./bdeckq  # Builds new Abaqus input deck using nodes.dat
  rm -f opjob.temp
  ./wrconv  # Writes peak hoop stress. Adds one line each iteration
  ./wrshape # Stores nodes.dat for each iteration in znodes???.dat
  time2=$(date +"%T")" "$(date +"%Y-%m-%d")
  secs2=$(date +"%s")
  etime=$(echo "scale=2;($secs2-$secs1)/60.0" | bc)
  echo ""
  echo "Iteration start time  : $time1"
  echo "Iteration finish time : $time2"
  echo "Iteration elapsed time: $etime minutes"
done

time3=$(date +"%T")" "$(date +"%Y-%m-%d")
```

```
secs3=$(date +"%s")
etime=$(echo "scale=3;($secs3-$secs0)/3600.0" | bc)

echo ""
echo "Job iterations  : $maxitns"
echo "Job start time  : $time0"
echo "Job finish time : $time3"
echo "Job elapsed time: $etime hours"
echo ""
echo "Shape optimisation job completed."
echo ""
```

DEFENCE SCIENCE AND TECHNOLOGY GROUP DOCUMENT CONTROL DATA	1. DLM/CAVEAT (OF DOCUMENT)

2. TITLE	3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)
Conversion of DST Group Shape Optimisation Software for Increased Portability across Computing Platforms	Document: (U) Title: (U) Abstract: (U)

4. AUTHOR(S)	5. CORPORATE AUTHOR
Robert Kaye and Witold Waldman	Defence Science and Technology Group 506 Lorimer St Fishermans Bend Victoria 3207 Australia

6a. DST GROUP NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DST-Group-TR-3251	AR-016-589	Technical Report	May 2016

8. OBJECTIVE FOLDER ID	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
fAV1044425	AIR 07/283	OIC-ASI-DGTA	93	9

13. DST GROUP PUBLICATIONS REPOSITORY	14. RELEASE AUTHORITY
http://dspace.dsto.defence.gov.au/dspace/	Chief, Aerospace Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

Yes

18. RESEARCH LIBRARY THESAURUS

Finite element analysis, Shape optimisation, Stress concentration

19. ABSTRACT

The DST Group shape optimisation methodology is well established with several successful implementations to ADF aircraft involving repairs to crack-prone locations. The process involves adaptive reshaping of locally-concave boundaries so as to minimise a stress concentration by spreading the stress more evenly over a longer region of the boundary. DST Group in-house software is used in conjunction with a commercial finite element solver in an iterative manner to achieve this outcome. The prior DST Group software had been found to be dependent on the version of the commercial graphical user interface being used and the software was not readily adaptable to newer versions. The work reported here involves replacing some of the prior code so that the graphical user interface is not used in the process. This document includes a number of 2D and 3D example problems that were used to demonstrate the successful operation of the converted code. The main benefit of the new code is that the software can be ported to other computer hardware without any interaction with the installed graphical user interface, but it also enables a reduction in the use of commercial licenses and provides faster run times.